Proceedings

28th Annual Computer Security Applications Conference

ACSAC 2012

Proceedings

28th Annual Computer Security Applications Conference

Orlando, Florida, USA
3–7 December 2012

Sponsored by
Applied Computer Security Associates

The Association for Computing Machinery
2 Penn Plaza, Suite 701
New York, New York 10121-0701

ACM ISBN: 978-1-4503-1312-4

Editorial production by Christoph Schuba
Published by ACM, Inc. within the ACM International Conference Proceedings Series

Table of Contents

Technical Program

Web Security

Mobile Security

Hardware Security

Passwords

Botnets

Classic Book

Invited Essayist

Authentication

Code Analysis Techniques

Cloud Security

Intrusion Detection

Policy

Protection Mechanisms

Malware Analysis and Classification

Software Security

Social Networking Security

Systems Security

Message from the Conference Chair

On behalf of the Organizing Committee, I would like to welcome you to the 28^{th} Annual Computer Security Applications Conference. We have selected a diverse, exciting program this year, and I trust you will find several items of interest, whether you are attending in person or reading the proceedings. ACSAC is about more than just sitting through a presentation or reading a paper however, and I would encourage you to interact with your peers in the security field, and strive to improve the state of security in our world. Don't be afraid to challenge old ideas, suggest new ones, or improve upon the status quo. Use ACSAC as an opportunity to learn, participate, network, and make a difference!

Robert H'obbes' Zakon, ACSAC 2012 Conference Chair

Message from the Program Chairs

Welcome to the 28^{th} Annual Computer Security Applications Conference. Such a large conference represents the hard work of many talented volunteers, and the formulation of the technical track is no exception. From authors to Program Committee members, a technical program would not exist without the time and effort they spent in writing, editing, discussion, and reviewing.

First, all submitting authors deserve thanks for their efforts to communicate their latest research through ACSAC as a venue. The growing strength of ACSAC is in part derived from the interest and attention drawn by the quality of work submitted for review and accepted for publication. We received a total of 231 valid submissions this year – a record for ACSAC. After some submissions were rejected on formal grounds or retracted by their authors, the reviewing phase selected 44 papers (a 19% acceptance rate). Our mandate as a PC was to try to accept as many quality papers as possible, and we were able to do so, keeping pace with our increased submission numbers. Even so, we had to turn away many promising papers, and we wish those authors best of luck in refining their work.

Second, we are very grateful to our Program Committee. Reviewing is hard work, and the increased submission numbers meant an increased reviewing load. Part of this workload was borne by some external subreviewers, and we appreciate their efforts. Program Committee members were ultimately responsible for the reviews. After the review phase, the Program Committee met for online discussions using OpenConf during the first two weeks of August. We thank all the PC members who gave freely of their time to discuss their reviews, arrive at consensus, and in some cases provide extra reviews. They did an amazing amount of work in a short period of time. To our PC: **thank you!**

The resulting technical program contains cutting edge research in a applied security. One of the most enjoyable parts of the process for the Program Chairs was putting together the actual program sessions. It is exciting to see the best work emerge from the review phase and to identify trends and topics from the accepted papers. Our fifteen technical track sessions represent a broad and interesting snapshot of the field. We have papers on hot topics like mobile security, cloud security, and social network security along with sessions on "traditional" systems and applied security areas like code analysis, software security, intrusion detection, and authentication.

Our final thanks go to you – the ACSAC attendee and the reader of this proceedings. Your interest plays a vital role in making ACSAC an important part of creating and disseminating scientific research in information security. Enjoy what you are reading, and consider contributing to ACSAC in the future!

Michael Locasto, ACSAC 2012 Program Chair
Patrick Traynor, ACSAC 2012 Program Co-Chair

Organizing Committee

Robert H'obbes' Zakon, Zakon Group LLC (Conference Chair)
Michael Locasto, University of Calgary (Program Chair)
Patrick Traynor, Georgia Institute of Technology (Program Co-Chair)
Christoph Schuba, Oracle Corporation (Proceedings Chair)
Steve Rome (Case Studies Chair)
Larry Wagoner, NSA (Case Studies Co-Chair)
Marshall Abrams, The MITRE Corporation (FISMA Coordination Chair)
Kevin Butler, University of Oregon (Panels Chair)
Benjamin Kuperman, Oberlin College (Posters & Works-in-Progress Chair)
Daniel P. Faigin, The Aerospace Corporation (Professional Development Chair)
Harvey H. Rubinovitz, The MITRE Corporation (Workshops Chair)
Rance DeLong, LynuxWorks (Layered Assurance Workshop Chair)
Cristina Serban, AT&T Security Research Center (Invited Speakers Coordinator)
Dan Thomsen, SIFT (Knowledge Coordinator)
Ann Marmor-Squires, The Sq Group (Local Arrangements Coordinator)
Charles Payne, Adventium Labs (Publications Coordinator)
Raheem Beyah, Georgia Institute of Technology (Publicity Coordinator)
Yingfei Dong, University of Hawaii (Publicity Coordinator)
Art Friedman, NSA (Registration Coordinator)
Hassan Takabi, University of Pittsburgh (Sponsorship Coordinator)
Ben Cook, Sandia National Lab (Student Coordinator)
Ed Schneider, Institute for Defense Analyses (Treasurer)
Zed Abbadi, Public Company Accounting Oversight Board (Co-Treasurer)
Jay Kahn (ACSA Communications Chair)
Carrie Gates (ACSA Government Liaison)

Steering Committee

Marshall Abrams, The MITRE Corporation
Jeremy Epstein, SRI International
Daniel P. Faigin, The Aerospace Corporation
Carrie Gates, CA Labs
Ann Marmor-Squires, The Sq Group
Steve Rome
Ron Ross, National Institute of Standards and Technology
Christoph Schuba, Oracle Corporation
Cristina Serban, AT&T Security Research Center
Dan Thomsen, SIFT

Program Committee

Michael Locasto, University of Calgary (Program Chair)
Patrick Traynor, Georgia Institute of Technology (Program Co-Chair)

John Aycock, University of Calgary
Lucas Ballard, Google, Inc.
Darrell Bethea, University of North Carolina at Chapel Hill
Kevin Butler, University of Oregon
Justin Cappos, NYU Poly
Hao Chen, University of California, Davis
Mihai Christodorescu, IBM T.J. Watson Research Center
Gabriela Ciocarlie, SRI International
Christian Collberg, University of Arizona
Jedidiah Crandall, University of New Mexico
Weidong Cui, Microsoft Research
Sven Dietrich, Stevens Institute of Technology
Yingfei Dong, University of Hawaii
Richard Ford, Florida Tech
Michael Franz, University of California, Irvine
Carrie Gates, CA Technologies
Phillipa Gill, University of Toronto
Steven Greenwald, Independent Consultant
Guofei Gu, Texas A&M University
Sean Heelan, Immunity Inc.
Trent Jaeger, Penn State University
Christopher Masone, Google, Inc.
John McDermott, Naval Research Laboratory
Jelena Mirkovic, USC/ISI
Thomas Moyer, MIT Lincoln Laboratory
Steven Myers, Indiana University
Jon Oberheide, Duo Security
Bryan Payne, Nebula, Inc.
Michalis Polychronakis, Columbia University
Georgios Portokalidis, Columbia University
Pierangela Samarati, Universitá degli Studi di Milano
Christoph Schuba, Oracle Corp.
Cristina Serban, AT&T Security Research Center
Gaurav Shah, Google, Inc.
Micah Sherr, Georgetown University
Sara Sinclair Brody, Google, Inc.
Kapil Singh, IBM T.J. Watson Research Center
Anil Somayaji, Carleton University
Abhinav Srivastava, AT&T Labs
Angelos Stavrou, George Mason University
Julie Thorpe, University of Ontario Institute of Technology
Laura Tinnel, SRI International
A. Selcuk Uluagac, Georgia Institute of Technology
Larry Wagoner, NSA
Gaven Watson, University of Bristol
John Wilander, Svenska Handelsbanken
Dongyan Xu, Purdue University

External Reviewers

Sadia Akhter, Chaitrali Amrutkar, Claudio A. Ardagna, David Barrera, David Bernhard, Stefan Brunthaler, Jean Camp, Hank Carter, Dana Dachman-Soled, Sabrina De Capitani di Vimercati, Daniel de Castro, Zhui Deng, Will Enck, Philip W. L. Fong, Sara Foresti, Georg Fuchsbauer, Vinod Ganapathy, Carl Gebhardt, Ashish Gehani, Eleni Gessiou, Zhongshu Gu, Eric Hennigan, Andrei Homescu, Nathaniel Husted, Faisal Iqbal, Markus Jakobsson, Ravi Jhawar, Joshua Joy, Per Larsen, Chaz Lever, Dawei Li, Tao Li, Xiaodong Lin, Enchang Liu, Giovanni Livraga, Long Lu, Shoufu Luo, Liam Mayron, Monzur Muhammad, Divya Muthukumaran, Terri Oda, Stefano Paraboschi, Kenneth Paterson, Roberto Perdisci, Amirali Salehi-Abari, Sumanta Sarkar, Seungwon Shin, Yingbo Song, Mudhakar Srivatsa, Dannie Stanley, Yuqiong Sun, Nirupama Talele, Henry Tan, Sai Teja Peddinti, Hayawardh Vijayakumar, Chris Wacek, Ting Wang, Glenn Wurster, Zhaoyan Xu, Chao Yang, Vinod Yegneswaran, Jialong Zhang

Professional Development Course Proposal Reviewers

Daniel P. Faigin, The Aerospace Corporation (Professional Development Chair)
Colin Cole, The Aerospace Corporation
Patricia Daggett, EMC Corporation
Ryan Eads, Sallie Mae
John Mildner, SPAWAR Systems Center Atlantic
Donna Mitchell, Lockheed Martin Corporation
Fiona Pattinson, atsec information security
W. Warren Pearce, Retired
Scott Perry, SecureInfo - A Kratos Company
Michelle Ruppel, Saffire Systems
Harvey Rubinovitz, The MITRE Corporation
Cristina Serban, AT&T Security Research Center
Danielle Weigold, Lockheed Martin Corporation
Simon Wiseman, Deep-Secure Ltd

ACSA Members

Marshall Abrams, The MITRE Corporation (ACSA Founder and Assistant Treasurer)
Jeremy Epstein, SRI International (ACSA President)
Daniel P. Faigin, The Aerospace Corporation (ACSA Secretary)
Carrie Gates, CA Technologies
Ann Marmor-Squires, The Sq Group (ACSA Vice President)
Harvey Rubinovitz, The MITRE Corporation (ACSA Treasurer)
Cristina Serban, AT&T Security Research Center
Mary Ellen Zurko, Cisco Systems

Message from the Sponsor

Applied Computer Security Associates

ACSA had its genesis in the first Aerospace Computer Security Applications Conference in 1985. That conference was a success and evolved into the Annual Computer Security Applications Conference (ACSAC). ACSA was incorporated in 1987 as a non-profit association of computer security professionals who have a common goal of improving the understanding, theory, and practice of computer security. ACSA continues to be the primary sponsor of the annual conference.

In 1989, ACSA began the Distinguished Practitioner Series at the annual conference. Each year, an outstanding computer security professional is invited to present a lecture of current topical interest to the security community. In 1991, ACSAC began the Best Paper by a Student Award, presented at the Annual conference. This award is intended to encourage active student participation in the conference. The award winning student author receives an honorarium and conference expenses. Additionally, our Student Conferenceship program assists selected students in attending the Conference by paying for the conference fee. Applicants must be undergraduate or graduate students, nominated by a faculty member at an accredited university or school, and show the need for financial assistance to attend this conference.

An annual prize for the Outstanding Paper has been established for the Annual Computer Security Applications Conference. The winning author receives a plaque and an honorarium. The award is based on both the written and oral presentations. ACSA initiated the Marshall D. Abrams Invited Essay in 2000 to stimulate development of provocative and stimulating reading material for students of Information Security, thereby forming a set of Invited Essays. Each year's Invited Essay addresses an important topic in Information Security not adequately covered by the existing literature. This year's ACSAC continues the Classic Papers feature begun in 2001. The classic papers are updates of some of the seminal works in the field of Information Security that reflect developments in the research community and industry since their original publication. Each presentation also considers how these classical security results will impact us in the years to come.

ACSA continues to be committed to serving the security community by finding additional approaches for encouraging and facilitating dialogue and technical interchange. In previous years, ACSA has sponsored small workshops to explore various topics in Computer Security (in 2000, the Workshop on Innovations in Strong Access Control; in 2001, the Workshop on Information Security System Rating and Ranking; in 2002, the Workshop on Application of Engineering Principles to System Security Design). In 2003, ACSA became the sponsor of the already established New Security Paradigms Workshop (NSPW). In 2010 we welcomed the Layered Assurance Workshop as an affiliated ACSA activity. In 2012, ACSA co-founded the Learning from Authoritative Security Experiment Results (LASER) workshop, which is expected to become an annual event. ACSA also maintains a Classic Papers Bookshelf that preserves seminal works in the field and a web site focusing on Strong Access Control/Multi- Level Security. After 2010's ACSAC tribute to Paul Karger and Bob Abbott, ACSA has initiated work on an In Memoriam section of our website to capture the many contributions made by our fellow practitioners; in 2011 we added Gene Schultz and H.O. Lubbes; and in 2012 we bit farewell to William Winsborough and Hal Tipton.

Building for the future, in 2011 ACSA started the Scholarships for Women Studying Information Security (SWSIS) program to encourage young women to join the field; in 2012 we awarded $25,000 in scholarships, and anticipate continuing to offer scholarships in the future. Our winners are at ACSAC this year; please congratulate them and welcome them to our field! ACSA is always interested in suggestions from interested professionals and computer security professional organizations on other ways to achieve its objectives of encouraging and facilitating dialogue and technical interchange. For more information on ACSA and its activities, please visit http://www.acsac.org/acsa

Contributing Sponsors

JSand: Complete Client-Side Sandboxing of Third-Party JavaScript without Browser Modifications

Pieter Agten[†], Steven Van Acker[†], Yoran Brondsema[†], Phu H. Phung[‡],
Lieven Desmet[†], Frank Piessens[†]
{pieter.agten,steven.vanacker}@cs.kuleuven.be

[†]IBBT-DistriNet, KU Leuven, 3001 Leuven, Belgium
[‡]ProSec Security Group, Chalmers University of Technology, Gothenburg, Sweden

ABSTRACT

The inclusion of third-party scripts in web pages is a common practice. A recent study has shown that more than half of the Alexa top 10 000 sites include scripts from more than 5 different origins. However, such script inclusions carry risks, as the included scripts operate with the privileges of the including website.

We propose JSand, a server-driven but client-side JavaScript sandboxing framework. JSand requires *no browser modifications*: the sandboxing framework is implemented in JavaScript and is delivered to the browser by the websites that use it. Enforcement is done *entirely at the client side*: JSand enforces a server-specified policy on included scripts without requiring server-side filtering or rewriting of scripts. Most importantly, JSand is *complete*: access to all resources is mediated by the sandbox.

We describe the design and implementation of JSand, and we show that it is secure, backwards compatible, and that it performs sufficiently well.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection; H.3.5 [**Information Storage and Retrieval**]: Web-based services

Keywords

Web Application Security, Web Mashups, Script Inclusion, Sandbox, Security Architecture

1. INTRODUCTION

In the last decade, the web platform has become the number one platform on the Internet. There is a clear paradigm shift from desktop applications and proprietary client-server solutions towards web-enabled services. An important catalyst for this paradigm shift has been the power of JavaScript as well as the advent of HTML5, giving web developers the tools to build rich and interactive websites.

As a consequence of this enormous growth in popularity, the web has also become the primary attack platform: SANS [30] reported in 2009 that more than 60% of all cyber attacks are aimed at web applications, and more than 80% of discovered vulnerabilities are web-related. A whole range of web attacks exists in the wild, ranging from Cross-Site Scripting, Cross-Site Request Forgery and SQL injection to the exploitation of broken authorization and session management. This paper focuses on one particular and important class of web attacks, namely attacks due to the insecure integration of JavaScript.

To enrich the functionality and interactivity of a website, a common and wide-spread approach is to integrate JavaScript from third-party script providers. Recent studies [40, 23] have shown that 96.9% of websites include scripts from external sources, and on average each website includes scripts from 3.1 external sources. For example, websites integrate among others JavaScript-enabled advertisements (such as Google AdSense and adBrite), Web analytics frameworks (such as Google Analytics, Yahoo! Web Analytics and Tynt), web widgets and buttons (such as Google Maps, addToAny button and Google +1 button), and JavaScript programming libraries (such as jQuery and Dojo).

The de facto browser security model today is defined by the Same-Origin Policy (SOP). The SOP restricts access of client-side scripts to resources belonging to the same origin[1]. For instance, the SOP ensures that document data and cookies from one origin cannot be read by scripts belonging to another origin. However, the SOP includes some important relaxations with respect to navigation and content inclusion (e.g. embedded images and scripts) [41]. In particular, if a page from one origin includes a script from another origin, the included script is treated as if it belongs to the including origin, and hence it inherits all the capabilities and permissions of the hosting page. This makes malicious script inclusion a very powerful attack vector.

Several countermeasures have been proposed to limit the capabilities of third-party JavaScript, including (1) the introduction of safe subsets of JavaScript [33, 3, 15], (2) client-side reference monitors [17, 34], and (3) server-side transformations of the script to be included [22, 32]. However, all of these have at least one of the following limitations.

First, some approaches [17, 34] require intrusive *browser modifications*, in particular to the JavaScript engine and the

[1]An origin is a (protocol, domain name, port) tuple.

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

binding between browser and JavaScript engine. Such modifications hinder short-term deployment of the countermeasure.

Second, some approaches do *not support client-side script inclusion*: in order to perform server-side pre-processing (e.g. source-to-source translation or filtering) of the scripts, the scripts have to pass through the web server [22, 33, 3]. This effectively changes the architectural model of client-side script inclusion to server-side script inclusion.

Third, some approaches do *not provide complete mediation* between different scripts on the same page, or to all resources exposed in the browser. Self-Protecting JavaScript (SPJS) [26, 16] assumes that all scripts included on a hosting page need identical security constraints. It does not differentiate between different external scripts nor between local and remote inclusions. AdJail [32] successfully isolates untrusted advertisements from the Document Object Model (DOM) of the hosting page, but since it uses iframes as isolation units, it cannot fully protect security-sensitive APIs such as XHR, Geolocation and local storage.

Inspired by recent advances in achieving object-capability guarantees for JavaScript [22, 14, 20, 25, 8], this paper presents JSand, a novel security architecture to securely integrate third-party JavaScript. We improve upon the state-of-the-art with the following contributions:

1. JSand is the first JavaScript sandbox that (1) does not need browser modifications, (2) supports client-side script inclusion and (3) completely mediates different scripts and the browser APIs.

2. We show evidence that JSand is secure, compatible with complex and widely used scripts (such as Google Maps, Google Analytics and jQuery) and performs sufficiently well.

The rest of this paper is structured as follows. Section 2 introduces the necessary background and defines the problem statement. In Section 3, the JSand architecture is presented, and Section 4 discusses several relevant implementation aspects. Section 5 evaluates the security, compatibility and performance of JSand. Finally, Section 6 discusses related work, and we conclude in Section 7.

2. PROBLEM STATEMENT

2.1 Integrating third-party JavaScript

To enrich the functionality and interactivity of a website, a common and wide-spread approach is to integrate JavaScript from third-party script providers. The two most wide-spread techniques to integrate third-party JavaScript in web pages are through script inclusion and via iframe integration [6].

Script inclusion HTML script tags are used to execute JavaScript as part of a web page. If the JavaScript code is integrated from an external source, the browser will still execute the code within the security context of the web page, without any restrictions of the SOP.

Iframe integration HTML iframe tags allow a web developer to include one document inside another. The advantage of iframe integration is that the integrated document is loaded in its own security context: integrated content from another origin is isolated from the integrating web page by the SOP.

Script inclusion is the de facto script integration technique on the web, both for local scripts as well as for external scripts. The iframe integration technique is used for web gadgets that don't have strong integration needs with the embedding web page, or have an out-of-band service-to-service communication channel (such as the Facebook Like button or Facebook Apps). In the remainder of this paper, we focus on third-party JavaScript integration through script inclusion.

2.2 Malicious script inclusion

The browser security model for integrating third-party JavaScript is problematic. Once included in a website, a malicious script cannot only access all the document data and cookies, but with the advent of HTML5, the malicious script has also access to local storage data (e.g. Web Storage, IndexedDB), intra-window communication (Web Messaging), remote resource fetching via XHR and user-consented privileges (such as Geolocation, media capture, access to System Information API, and many more). This makes malicious script inclusion a very powerful attack vector. One can distinguish between two types of attackers.

Malicious script provider The script provider has malicious intentions (but covers up by providing appealing functionality to potential customers), or becomes malicious over time (e.g. intentionally, or by selling out or quitting his business [23]).

Benign script provider under attack The script provider has no malicious intentions, but the scripts delivered to its clients become under control of an attacker. This can be due to the inclusion of other untrusted resources (e.g. in advertisement networks), due to a bug in the delivered script (e.g. a DOM-based XSS vulnerability [12]), due to a server-side take-over (e.g. via SQL injection) or due to in-transit tampering with the scripts by a network attacker.

In both cases, the attacker controls the scripts included by the hosting page, and by default gains full access to the execution environment of the web page.

2.3 Requirements

Given the wide spread of script inclusion and the increasing impact of malicious script inclusion, there is a clear need for a novel security architecture to **securely integrate third-party JavaScript**, but **without introducing disruptive changes**. Preserving backwards compatibility is crucial in the web context. We therefore identify the following requirements:

R1 Complete mediation All access to security-sensitive functionality should be completely mediated by the security mechanism. This includes access to the DOM, as well as security-sensitive JavaScript APIs (such as Geolocation and local storage). The attacker must be unable to circumvent the security mechanisms in place.

R2 Backwards compatible The security mechanism should seamlessly operate in the current web ecosystem, i.e. it should not rely on browser modifications or disable the direct delivery of scripts from the script provider to the browser. In addition, the security mechanism should support the integration of legacy scripts.

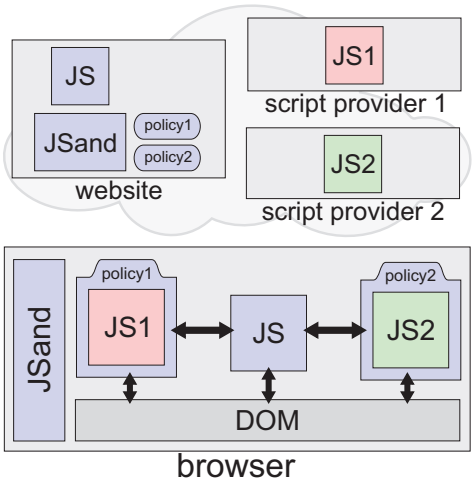


Figure 1: the JSand architecture. Inside the browser, all access from JSand sandboxes to the JavaScript environment is mediated according to server-supplied policies.

R3 Performance The security mechanism should introduce only a minimal performance penalty, unnoticeable to the end-user.

3. JSAND SECURITY ARCHITECTURE

The JSand architecture enables the owner of a website to securely integrate third-party scripts, without needing disruptive changes to either server-side or client-side infrastructure. We first give a high-level overview of the architecture and then discuss the architectural choices under the hood.

3.1 Architectural overview

Figure 1 depicts the JSand architecture. A website owner deploys JSand by including the JSand JavaScript library in his webpages. When one of these pages is loaded in a visitor's browser, the third-party scripts to be sandboxed are fetched directly from the servers of the script provider. The JSand library confines each script to its own secure sandbox, which isolates the script from other scripts and from the DOM.

3.2 Under the hood

The JSand architecture is based upon the secure confinement of third-party JavaScript. JSand realizes this through the use of an object-capability environment. Such environment provides an appropriate device for isolating untrusted JavaScript: without an explicit and unforgeable reference to a security-sensitive resource (i.e. an object or a function), a script is unable to access the resource or make use of its capabilities. The object-capability model is at the basis of Caja [22], and many other safe subsets of JavaScript [14].

The JSand library invokes third-party scripts, initially giving them only a minimal set of unforgeable references. To maintain control over all references acquired by a sandboxed script, JSand applies the Membrane pattern proposed by Miller [21]. Our implementation of this pattern consists of placing policy-enforcing wrappers around objects that provide potentially security-sensitive operations. Whenever one of these objects returns a reference to another object, the membrane is extended to cover that object as well. This ensures a sandboxed script never has direct access to a security-sensitive operation.

The membrane's wrappers intercept all operations performed on the objects they wrap and hence implement the security policy enforcement points. On each enforcement point, the wrapper consults the security policy to determine whether or not the corresponding operation is permitted. If not, this will be indicated by the security policy and the operation will be blocked. The architecture is not bound to any specific type of security policy, which gives website owners the freedom to enforce arbitrarily complex policies.

Since all interactions between a script and the browser are performed by calling DOM methods, it suffices to place a wrapper around each DOM object in order to enforce a policy on all security-sensitive operations. These include not only operations to read or modify content of the hosting page, but also to communicate with other scripts and to use browser-provided JavaScript APIs.

In conclusion, the JSand architecture provides an end-to-end solution for securely integrating third-party JavaScript scripts on a website. The website owner is able to define and enforce security policies on scripts, which puts him back into the driver's seat. JSand does not require disruptive changes to the architecture of the web: it does not break direct script delivery towards the browser, and can be deployed without additional server-side or client-side infrastructure. The combination of the object-capability model and the Membrane pattern ensures that all access from a sandboxed script to security-sensitive operations passes through a membrane's wrappers, which enforce the security policy.

4. PROTOTYPE IMPLEMENTATION

This section reports on the development of a mature JSand prototype, which is designed to work in ECMAScript 5 (ES5) compatible browsers with support for the proxy features of the upcoming ES Harmony standard. The current prototype runs seamlessly in Google Chrome v20.0.1132.21.

In Subsections 4.1 and 4.2, we present the client-side technology for executing third-party JavaScript in a confined sandbox. Subsection 4.3 describes the type of security policies that are enforced. Next, Subsection 4.4 illustrates how access to security-sensitive operations is completely mediated. Subsection 4.5 discusses how our prototype deals with dynamic script loading and Subsection 4.6 describes a set of automatic script transformations to improve compatibility with legacy scripts.

4.1 Object-capability system

As described in Section 3.2, the JSand architecture relies on an object-capability environment to provide complete mediation. The ECMAScript language does not qualify as an object-capability language by itself. For instance, any script has access to all global variables by default, and consequently has capabilities that are not under control of any security framework. However, in 2008, Miller et al. [22] identified a subset of ES3 which forms a true object-capability language. More recently, the Google Caja team has identified a subset of ES5 strict, named Secure ECMAScript (SES), that also provides such an object-capability language. Moreover, they have developed a JavaScript library that enables the execution of SES on ES5-compatible browsers [20]. This library provides methods for safely evaluating SES-compliant

code in an isolated environment. A key feature of the library is that it can execute completely at the client side and hence doesn't rely on any custom server-side architecture. JSand uses the SES library to realize its underlying object-capability environment.

However, since SES is a subset of ES5 strict, which in turn is a subset of ES5 non-strict, not all currently deployed JavaScript scripts are SES-compliant. Furthermore, the language supported by the SES library differs from true SES in several minor ways, further reducing compatibility with legacy scripts. Two important incompatibilities between ES5 and the SES-like language supported by the SES library are described below.

Global variables In ES5, the global `window` object can have arbitrary properties and for each of these properties there is a corresponding global variable with the same name. Conversely, for any global variable, a corresponding property with the same name is defined on the global object. In SES, this is no longer the case: global variables are not aliased by properties on the global object or vice versa.

Strict mode SES enforces strict mode for all scripts. Hence, ES5 non-strict code might be incompatible with SES. For instance, strict mode drops support for the `with` keyword, prevents the introduction of new variables into the outer scope by an `eval` and no longer binds `this` to the global object in a function call.

SES was designed to support (only) recognized ES5 best practices. Therefore, scripts that adhere to these best practice standards are SES-compliant and hence we expect the number of fully SES-compliant scripts to increase progressively as these best practices become more widespread. Although not all legacy scripts run without errors under the SES library, the secure confinement of these scripts is never at stake. Nevertheless, we have developed a support layer to improve compatibility with legacy scripts. This layer is described in detail in Section 4.6.

To enforce the object-capability model and to provide support for legacy scripts, the SES library and the support layer need access to the source code of scripts to be sandboxed. Our prototype fetches this code using the XMLHttpRequest (XHR) API. By default, this API is subject to the SOP, but recently added web features have facilitated cross-domain interactions, namely Cross-Origin Resource Sharing (CORS) [37] and the Uniform Messaging Policy (UMP) [38]. In case CORS or UMP are not supported by the script provider, our solution can fall back to a server-side JavaScript proxy [39].

4.2 Policy-enforcing membranes

4.2.1 The Proxy API

To implement the Membrane pattern in an efficient and transparent way, our prototype uses the Harmony Proxy API, which is scheduled to be standardized in the next version of ECMAScript [35]. This API enables us to create wrappers that generically intercept all property accesses and assignments on specific objects, as shown in the code below.

```
function wrap(target, policy) {
  var handler = {
    get: function(proxy, propertyName) {
      if (policy.isGetAllowed(propertyName)) {
        return target[name];
      }
      return null;
    }
    set: function(proxy, propertyName, value) {
      if (policy.isSetAllowed(propertyName)) {
        target[name] = value;
        return true;
      }
      return false;
    }
  }
  return Proxy.create(handler, Object.getPrototypeOf(
      target));
}
```

The `wrap` function creates a simple policy-enforcing wrapper around a specific `target` object. All property accesses and assignments on this wrapper are intercepted by the `get` and `set` traps of the handler object, which uses the `policy` object to determine whether or not the access or assignment is allowed.

4.2.2 Membrane implementation

To implement the Membrane pattern, the handlers used in JSand transitively wrap all objects they return from the `get` trap and unwrap the objects they receive in the `set` trap. The entire prototype chain of a wrapper must be wrapped as well, to prevent an attacker from piercing the membrane by accessing an unwrapped prototype.

If an object to be returned from the `get` trap is a function, a function proxy that wraps the original function is returned. This function proxy first unwraps all its arguments, then calls the original function using these unwrapped arguments and finally wraps the return value before returning it to the caller, thereby further expanding the metaphorical membrane. Some methods, such as `window.addEventListener`, take a callback function as an argument; like all other arguments, this callback must be wrapped appropriately to uphold the membrane. Because a callback function is executed in the context of a sandbox, its wrapper must wrap each of its arguments and must unwrap the return value after calling the original function with the wrapped arguments.

Each sandbox keeps a mapping from its wrappers to the target objects they wrap and vice versa. This makes it possible to unwrap previously wrapped objects and to ensure that there is at most one wrapper (per sandbox) corresponding to each target object, making the membrane identity-preserving [4]. The mapping from wrappers to their corresponding targets is only accessible from outside the sandbox, for otherwise an attacker could use it to escape from the sandboxed environment.

Whereas sandboxed code should always be confined to the bounds of its own sandbox, many use cases require an operation to introduce code from outside a sandbox into an existing sandbox. Such operations enable a website owner to extend or interact with a sandboxed script. JSand sandboxes provide two functions for introducing new code into them: `innerEval(code)` and `innerLoadScript(url)`. The first function evaluates a literal code string, while the second loads a script at a given URL.

In conclusion, the Membrane pattern transparently isolates a sandbox from code running outside of it or in other sandboxes. Since the handlers intercept each property access and assignment made on a wrapper, they contain the enforcement points which consult the security policy to determine whether or not an operation is permitted.

4.3 Security policies

Defining good security policies is important for ensuring the secure confinement of sandboxed scripts. To avoid needing a *known-good* version of a script to be sandboxed, a policy should be based on the claimed functionality of a script, as opposed to being based on actions performed by any specific version of the script. Generic templates can be provided to support website owners in defining good security policies.

Since the JSand architecture is independent of the specific type of security policy to enforce, policies can range from simple stateless policies, to arbitrarily complex policies. In both cases, the security policy can be specified as a JavaScript function that takes information about the operation to be performed as input and returns a boolean indicating whether or not the operation is allowed. We discuss three types of policies in more detail below.

4.3.1 Stateless policies

Stateless policies determine whether or not an operation is permitted based on information associated with that operation alone. For instance, a stateless security policy could specify that a specific function call performed on a specific object is only allowed when the value of the first argument is on a predefined whitelist.

WebJail [34] is an example of such a stateless policy for securely integrating third-party JavaScript. It classifies security-sensitive operations into nine categories, including DOM access, cookies, external communication, device access, etc., which can be permitted or blocked individually. A WebJail policy is based on a static whitelist of each of these categories, and could easily be implemented with JSand.

4.3.2 Stateful policies

Stateful policies can accumulate internal security state over multiple calls and use this global state as part of the policy, in addition to the local information made available on each operation request. For instance, a stateful security policy could specify that the use of XHR is allowed as long as no cookies have been read. This type of policy is more expressive than its stateless counterpart, but it is also more complex to specify and more prone to mistakes.

The shadow page in AdJail [32] is another example of internal state that could be accumulated over multiple calls. This page represents a ghost DOM, which is not directly rendered to the user, but allows an advertisement to execute various DOM operations in a confined environment.

4.3.3 Advanced policies

More complex policies can be used to enforce more advanced security properties, such as information flow security. One example of this is a set of policies to implement *noninterference* through *secure multi-execution* (SME) [7, 5]. For any script, SME can classify each input and each output channel as either H (high security, confidential) or L (low security, public). A script is noninterferent if its low-level outputs are not influenced by high-level inputs. Consider for instance the following script on a webserver at `mydomain.com`.

```
var cookies = document.cookie;
document.getElementById('some-img').src = 'http://
    attacker.com/img.jpg?c=' + escape(cookies);
```

The first line can be classified as H input, since cookie values are security sensitive. The second line can be classified as L output, since this triggers an HTTP request to a different domain. This program is interferent, because the low-level output statement at line 2 is clearly influenced by the high-level input statement at line 1.

Under secure multi-execution, a script is run multiple times, once for each security level. Outputs of a given security level are only generated in the execution belonging to that security level and inputs of a given security level are replaced by `undefined` in all executions of a lower level. Hence, high-level, security-sensitive input can never leak to low-level, public output channels, or even have an influence on them.

To multi-execute a script using JSand, that script must be executed once for the low security level and once for the high security level, each time in a different sandbox, with a different security policy. The low-level policy should disable all high-level inputs and ignore high-level outputs, while the high-level policy should simply ignore low-level outputs. Since each output statement is executed in only one of the executions, the net effect of a noninterferent script under secure multi-execution will be the same as the net effect of executing the same script without multi-execution.

4.4 Wrapping the DOM

All interactions between a script and the browser are performed through the DOM. Hence, to control access to all security-sensitive operations, JSand needs to control access to all facets of the DOM. To implement this, each sandboxed script is initially only given a single reference to a wrapper of the `window` object, which is the root of the DOM tree. As described in Section 4.2, all property accesses, property assignments and function calls on this wrapper or on any object transitively reached from it are intercepted by a handler. These handlers can thus enforce an arbitrary policy on the entire DOM, and hence effectively control access to all security-sensitive operations.

For any DOM object wrapper, a distinction can be made between two categories of properties. The first category consists of *standard DOM properties*, i.e. properties that are part of the DOM as defined by the ECMAScript standard (or implementation-specific properties provided by the browser). The second category consists of *custom properties* that have been added to a DOM object wrapper by a sandboxed script. For instance, `window.document` belongs to the first category, while `window.googlemaps` could belong to the second. Properties from these two categories need to be handled differently. Assignments to standard DOM properties should be propagated outside the sandbox to the corresponding target property on the real DOM object (if allowed by the security policy), since this is the only way a sandboxed script can interact with the browser. Custom properties should however be confined to the bounds of the sandbox, to prevent sandboxed code from polluting the global namespace and from reading or modifying properties defined outside the sandbox.

To make the distinction between standard DOM properties and custom properties, JSand uses a statically defined *DOM description*, derived from the W3C DOM specification [36]. This description consists of an array of property descriptors, indexed by a DOM object name (e.g. `Window`) and a property name (e.g. `alert`). Since each descriptor corresponds to a standard DOM property, they enable the handlers to determine whether or not a given property is a standard DOM property

4.5 Dynamic script loading support

From experience, we have learned that many scripts dynamically load additional scripts during their execution. This is typically accomplished by inserting a new script tag with a `src` attribute in the document, because this method is not under restriction of the same origin policy. However, when a script is included this way, it is executed in the global context. Hence, if we would allow sandboxed scripts to simply add new script tags to the document, they could trivially break out of their sandbox. Any script included by a sandboxed script should execute within that same sandbox.

For this reason, JSand uses special handlers to intercept methods that allow script tags to be added to the document, including `node.appendChild`, `node.insertBefore`, `node.replaceChild`, `node.insertAfter` and `document.write`. The first four of these take a (partial) DOM tree as argument and append or insert it at a certain place in the DOM. Our handlers for these methods search the given DOM tree for script tags, extract the value of the `src` attribute and execute the corresponding scripts in the sandbox that included them, using the `innerLoadScript` function described in Section 4.2.2. The `document.write` method is similar but takes an HTML string as argument and appends that string verbatim to the document. The handler for this method parses the given HTML string, extracts script tags out of it and loads them as described above.

We have considered two different techniques for parsing a given HTML string in JavaScript. The first technique consists of creating an iframe and setting its `srcdoc` attribute [1] to the given HTML. To prevent the iframe from fetching and executing scripts included in the HTML, its `sandbox` attribute [1] must be set as well. The second technique consists of using a pure JavaScript library to parse the HTML [11]. The iframe-based technique has a potential performance benefit, since the parsing is done by native code in the browser instead of in JavaScript. Moreover, using the first technique ensures that the HTML is parsed exactly as the browser will interpret it. However, one of the problems of this approach, is that the parsing is performed asynchronously. That is, we can only access the iframe's fully populated DOM tree from its `onload` callback, which is triggered some time after setting the `srcdoc` attribute. Consequently, scripts that immediately make use of the HTML written by `document.write` could fail, since the HTML might not yet have been processed. Performing a continuation-passing style transformation on these scripts could solve this problem, but this is a complex transformation which we leave for future work. Our prototype uses the second technique, since it does not suffer from this problem.

4.6 Support for legacy scripts

Although the SES library natively supports scripts adhering to recognized ES5 best practices, as described in Section 4.1 not all currently deployed JavaScript scripts do so. Although the secure confinement of legacy scripts is never at stake, not all of them run without errors under the SES library. Therefore, we have developed a support layer to further improve the compatibility with these legacy scripts, based on three abstract syntax tree (AST) transformations.

T1 Adding a property to the global `window` object normally introduces that property as a global variable, but this does not hold in a SES environment. This transformation introduces a global alias variable for each property of `window`. The variable is updated whenever an assignment is made to its corresponding property.

T2 Conversely, declaring a global variable normally creates an alias property on the `window` object, but this doesn't hold in a SES environment. This transformation adds a property on `window` for each global variable. The property is updated whenever an assignment is made to its corresponding global variable.

T3 Since SES enforces strict mode for all scripts, ES5 non-strict code might be incompatible with SES. The most common incompatibility we have encountered is the lack of `this`-coercion. That is, `this` is no longer bound to the global `window` object in a function call. This transformation replaces `this` by the expression `(this === undefined ? window : this)`.

We have implemented a client-side component for applying these transformations, using the UglifyJS JavaScript parser [19]. These transformations do not provide a full translation from ES5 to SES, but they are sufficient to make many legacy scripts work with our prototype.

5. EVALUATION

In this section we evaluate to what extent JSand satisfies the requirements set forth in Section 2.3.

5.1 Complete mediation

All sandboxed scripts are executed in an object-capability environment, set up by the SES library. Our implementation of the Membrane pattern ensures that each DOM access and JavaScript API call made by a sandboxed script is assessed by the security policy. Based on the theory of object-capability systems, this provides complete mediation.

Note that JSand provides a one-way isolation and hence makes no attempt to protect a sandboxed script from its environment. That is, code running in the global security context, such as browser plugins and unsandboxed scripts, have the power to modify a sandbox's security policy or to inject a DOM proxy that allows access to any DOM object. However, since malicious global code has already full power over the web page, we consider protecting against such scenarios out of scope for our solution.

5.2 Backwards compatibility

We have extensively and successfully tested our prototype on a variety of JavaScript scripts. In this section we report and discuss in detail three of the most widespread included scripts around: Google Analytics, Google Maps and the jQuery library. Google Analytics is included from more than 68% of all domains from the Alexa Top 10 000, making it the most included script on this list [23]. Google Maps is the most included web mashup API according to [28], being used in 17.41% of registered mashups. jQuery is the most popular JavaScript library in use today, included in more than 57% of the top 10 000 websites to date [2]. As future work, we would like to extend our evaluation to more legacy scripts.

5.2.1 Google Analytics

Google Analytics (GA) is a web analytics service that generates statistics about visitors to a website. The GA API

allows web administrators to collect custom visitor properties, in addition to the standard properties that are collected by default (such as referrer and geographical location). The collected statistics can be monitored using a dashboard interface on the GA website.

To enable GA, the website owner must add a small JavaScript code template provided by Google to the header of the page to track. This template sets up an array of options to pass to the GA service and dynamically adds a new script tag to the page to include the main GA script. Any script included like this has unrestricted access to the DOM, making the page vulnerable to malicious script inclusions.

Manual inspection of the GA script is practically impossible, since the code is minified. Moreover, since the main GA script is loaded dynamically from the Google servers, any static, offline security analysis would fail to detect malicious changes introduced to the script after the initial analysis. However, by running GA in a JSand sandbox with a policy that permits only the operations necessary for a benign web analytics script, the impact of a malicious action on behalf of the GA script can be reduced to a minimum. The code snippet below shows how this can be implemented.

```
// main page:
var sb = new jsand.Sandbox('ganalytics.js',policy);
sb.load();
```

This code snippet creates a new sandbox and initializes it with the `ganalytics.js` script, which is shown below and consists of the code template provided by Google.

```
// ganalytics.js:
var _gaq = _gaq || [];
_gaq.push(['_setAccount', 'UA-xxxxxxxx-x']);
_gaq.push(['_trackPageview']);

(function() {
  var ga = document.createElement('script');
  ga.src = 'http://www.google-analytics.com/ga.js';
  var s = document.getElementsByTagName('script')[0];
  s.parentNode.insertBefore(ga, s);
})();
```

Both the `ganalytics.js` script and the main `ga.js` script (which is loaded from the code above) are executed in the same sandbox and are patched-up automatically, based on the AST transformations described in Section 4.6. The following code fragment shows the first two lines of the patched-up `ganalytics.js`.

```
// patched-up ganalytics.js:
var _gaq = _gaq || [];
window._gaq = _gaq;
[...]
```

The global variable `_gaq` is explicitly aliased as a property on `window`. This transformation is necessary because the `ga.js` script frequently refers to the `_gaq` array as `window._gaq`. Such references would fail without the patch shown here.

The `_gaq` array exposes an API to interact with GA after it has been initialized, for instance to add a custom property to collect or to track the click of a button. The website owner can access this array using the `innerEval` method described in Section 4.2.2. To facilitate these interactions and to make abstraction of the fact that GA is running in a sandbox, the website owner could implement an object that automatically forwards its calls to the `_gaq` array inside the sandbox.

Clearly, the effort required to run GA in a JSand sandbox is minimal and introduces no disruptive changes whatsoever. Nevertheless, the power of the GA script is reduced to a safe minimum, dramatically reducing the impact of a malicious script inclusion attack.

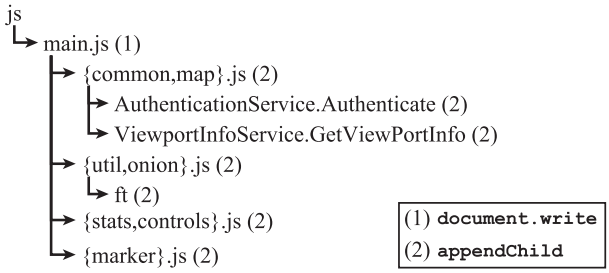


Figure 2: Tree of scripts dynamically loaded by Google Maps.

5.2.2 *Google Maps*

The Google Maps (GM) API enables website owners to embed a Google Maps gadget on their website. The standard way to add this gadget to a page is to (1) place a div element somewhere in the body where the map should be displayed, (2) add a script tag to the head of the page, which loads the GM library from the Google servers and (3) add a small piece of JavaScript code to the page, to create a new GM instance in the div element.

As with Google Analytics, the default way of including the GM script lets it have unrestricted access to the DOM and JavaScript APIs, putting the confidentiality and integrity of the entire web page at risk. JSand enables the website owner to confine the GM gadget to a sandbox with the minimal privileges required for legitimate operation.

The steps required to run GM in a JSand sandbox are very similar to the standard steps described above. In step (1), in addition to placing a div element somewhere in the body, the integrator must include the JSand library and the libraries it depends on. In step (2), instead of adding a script tag to directly load the GM library in the global page context, a new sandbox must be created for the GM script to run in. In step (3), the website owner can use the `innerEval` method to create a new GM instance in the sandbox. These steps are depicted in the following code fragment.

```
var sb = new jsand.Sandbox('http://maps.googleapis.
    com/maps/api/js?sensor=false', policy);
sb.load();
sb.innerEval(
 "var m = window.google.maps;
  var options = {
    center: new m.LatLng(-34.397, 150.644),
    zoom: 8, mapTypeId: m.MapTypeId.ROADMAP
  };
  var map = new m.Map(document.getElementById('
      map_div'), options);"
);
```

When the main GM script is loaded, a complex process of dynamically loading and patching other scripts is performed in the background. Figure 2 depicts the sequence of scripts that are dynamically loaded from the main `js` script initially loaded in step (2). In addition to the scripts shown in this figure, more scripts are loaded and patched whenever the user changes the map's viewport (by dragging it or changing the zoom level). All three translations described in Section 4.6 are required for the GM gadget to work.

The GM API provides extensive support for customization, to support feature-rich web mashups built around the

GM gadget. For instance, website owners can provide custom map overlays, place markers, register callbacks for mouse events, etc. As with GA, a website owner can use the `innerEval` method to interact with the sandboxed GM gadget.

The fact that JSand can successfully execute this gadget in a sandbox without any compatibility issues, illustrates that our solution is able to sandbox complex JavaScript gadgets that depend on dynamic script inclusions and that feature advanced DOM interactions.

5.2.3 jQuery

The jQuery library aims to provide a simple cross-browser API for performing common JavaScript operations, such as creating and selecting DOM elements, handling events, invoking Ajax interactions, etc. While jQuery can be used as an abstraction layer on top of an extensive set of JavaScript APIs, a website owner typically uses only a limited subset of what the library has to offer. By running jQuery in a sandbox with tight restrictions on the permitted JavaScript API and DOM operations, the risk and impact of a malicious script inclusion attack are reduced dramatically.

For our jQuery evaluation scenario, we executed jQuery together with the jQuery-geolocation plugin [24] in a sandbox, using a fine-grained security policy that allows us to toggle access to the JavaScript Geolocation API. Disabling the Geolocation API in the policy effectively prevents jQuery from using it in the sandbox. The following code fragment shows how this scenario is implemented.

```
var sb = new jsand.Sandbox('jquery-1.7.2.js',policy);
sb.load();
sb.innerLoadScript('jquery-geolocation-0.1.js');
sb.innerEval(
 "if (jQuery.geolocation.support()) {
    jQuery.geolocation.find(function(loc) {
      alert(loc.latitude+\", \"+loc.longitude);
    });
  } else { alert('Geolocation not supported'); }");
```

This scenario illustrates that, with minimal effort, a website owner can create a secure JSand sandbox around an extensible JavaScript library, while still being able to interact with it from outside the sandbox.

5.3 Performance benchmarks

To evaluate the runtime overhead of our prototype, we have conducted micro- and macro-benchmarks. All benchmarks were run using Google Chrome v20.0.1132.21 on Ubuntu 11.04 x86-64, running on an Intel Core 2 Duo T8300 2.4GHz processor with 4 GiB of RAM.

5.3.1 Micro benchmarks

JSand framework load time. To measure the load time of the JSand framework, a page was created that loads the framework but doesn't use it. This page was reloaded 1000 times and the elapsed time was recorded. The average load time measured in this way was 71.5±1.8 ms. The same experiment was run with all JavaScript code commented out, so the same network load time would be maintained, but the code would not be executed. The load time in this case was 23.0±0.2 ms. This means that once loaded from the network, the framework takes on average 48.5 ms to deploy on the client side.

Third-party library load time. Similar experiments were performed to measure the overhead of loading and parsing a third-party JavaScript library into a JSand sandbox. We chose jQuery as a representative JavaScript library and loaded it in a JSand sandbox, as well as a regular, unsandboxed JavaScript environment, using XHR and eval(). In both cases, we supplied a real JavaScript library as well as a commented-out version to factor out network overhead.

In a regular JavaScript environment, the code loads in 53.0±0.8 ms and 26.8±0.2 ms for normal and commented-out code respectively. Inside a JSand sandbox, the code loads in 1458.2±16.0 ms and 107.6±1.4 ms respectively, so that the overhead of parsing the library code is about 1350.6 ms.

A large portion of this overhead is due to the script rewriter of the legacy support layer described in Section 4.6. Since jQuery is SES-compliant, this rewriting step is not required. Disabling it lowers the average load time from 1458.2 ms to 705.8±1.1 ms, and the average overhead from 1350.6 ms to 598.2 ms. This means that 44.3% of the overhead can be contributed to our efforts for making legacy code SES-compliant.

Membrane transition cost. To verify the runtime overhead of a function call crossing the membrane, a function was executed both inside and outside a JSand sandbox 1 million times and the elapsed time is recorded. We chose the `window.clearTimeout` function as a representative function, because intuitively it should return quickly when no timer is registered. When called from inside the sandbox, the `window.clearTimeout` call must cross the membrane separating the sandbox from the real JavaScript environment. Outside the sandbox, the average execution time is 0.9±0.0 μs, while inside the sandbox it is 8.0±0.1 μs.

5.3.2 Macro benchmarks

The most important metric that counts when executing JavaScript in a browser, is the user experience. Ideally, the user should not notice that JSand is being used at all. To measure how much overhead the user experiences, we created a typical web application using Google Maps and measured two things: the total load time of the web application, and the delay a user experiences when interacting with it.

The load time of the web application was measured from the time the page is loaded until the Google Maps API emits a 'tilesloaded' event, signaling that the application is ready to be used. Running outside of the JSand sandbox, this load time is 308.0±13.7 ms, and 1432.8±24.2 ms inside of it. Keeping in mind that a large portion of this overhead is due to script-rewriting for legacy code, the total overhead without the legacy support layer can be estimated to be about 626.5 ms.

To measure the delay experienced when interacting with the application, we waited until the application was loaded, and then panned 400 px to the right, 100 times. The average time elapsed between two pans was considered as a reasonable approximation of the user-experienced delay. This delay is 320.2±0.8 ms outside and 420.0±2.7 ms inside the sandbox.

The overall performance of a JSand sandbox is acceptable. The overhead when loading a reasonably-sized SES-compliant JavaScript library inside the sandbox, is about 203%. For legacy scripts, JSand requires a code transformation step that results in a total overhead of about 365%,

but it is expected that this step can be removed or at least sped up significantly for future JavaScript code in future browsers. Furthermore, the tendency of users to keep certain websites open using persistent tabs, makes the load time overhead less important. Additionally, despite the ninefold execution time of a function-call traversing the sandbox membrane, the delay experienced by a user when using a realistic web application inside a JSand sandbox, is an acceptable 31.2%, corresponding to an absolute delay on the order of 100 ms.

6. RELATED WORK

Server-side processing of scripts.

A common technique for preventing undesired script behavior is to restrict the untrusted code (i.e. the third-party component) to a safe subset of JavaScript [15]. Compliance to the subset is verified at the server side. The allowed operations within the subset prevent the untrusted code from obtaining elevated privileges, unless explicitly allowed by the integrator. ADSafe [3], ADsafety [27] and FBJS [33] are examples of techniques where third-party JavaScript must conform to a certain JavaScript subset. Techniques such as Caja [22], Jacaranda [9] and Live Labs' Websandbox [18] on the other hand, statically analyze and rewrite the third-party JavaScript on the server side into a safe version.

Instead of forcing the use of a JavaScript subset, the JavaScript code can also be instrumented with extra checks that mediate access to certain functionality. BrowserShield [29] and Browser-Enforced Embedded Policies (BEEP) [10] are examples of such instrumentation on the server-side.

While safe subsets, code rewriting and server-side code instrumentation can restrict third-party code at the source, their adoption by mashup integrators is problematic. These techniques require either access to code running on the server-side, or require the website owner to implicitly trust the JavaScript provider to deliver safe JavaScript code. In real-world scenarios, it is infeasible to impose any such restrictions on third-party code providers. In contrast, JSand requires no server-side processing of the third-party code and imposes no fundamental restrictions on included code.

Extending the browser with a reference monitor.

A second class of techniques extends the browser to enforce code restrictions. Systems like ConScript [17], WebJail [34] and Contego [13] require modifications to the JavaScript engine to enforce policies on third-party code, while AdSentry requires the installation of a Firefox extension to restrict the functionality available to advertisements.

Browser modifications to restrict third-party JavaScript can be implemented very efficiently and can guarantee that enforcement cannot be circumvented. The major disadvantage of this approach however, is that the browser must be modified. Unless all the users of a web application are using a browser which implements the desired modification, there is little or no incentive for the website owner to make use of it. Because of the large variety of active browser vendors and versions on the internet, it is unrealistic to assume that a certain modification will ever be implemented in all browsers. For this reason, JSand does not depend on any special browser-side features except for what is available in the web standards.

Leveraging existing browser security features.

Finally, some approaches leverage recent browser security extensions to contain scripts. The new `sandbox` attribute of the iframe element in HTML5 [1] can restrict third-party JavaScript in a very coarse-grained way: it only supports to completely enable or disable JavaScript.

The Content Security Policy (CSP) [31] allows the insertion of a security policy through HTTP response headers and meta tags, which must be enforced in the browser. This policy can restrict the locations a web application loads its content from, thus preventing some forms of content-injection. However, CSP does not provide any fine-grained control over which JavaScript functionality is available to a script.

AdJail [32] is geared towards securely isolating ads from a hosting page for confidentiality and integrity purposes, while maintaining usability. The ad is loaded on a shadow page that contains only those elements of the hosting page that the web developer wishes the ad to have access to, and it relies on the SOP to isolate the shadow page. Changes to the shadow page are replicated to the hosting page if those changes conform to a specified policy. Likewise, user actions on the hosting page are mimicked to the shadow page if allowed by the policy. AdJail is a good approach to restrict access to the DOM, but cannot enforce a policy on the other JavaScript APIs like JSand does.

Self-protecting JavaScript (SPJS) [26, 16] is a client-side wrapping technique that applies advice around JavaScript functions, without requiring browser modifications (unlike [17] or [34]). It builds on standard aspect-oriented libraries for JavaScript. The wrapping code and advice are provided by the server and are executed first, ensuring a clean environment to start from. SPJS does not guarantee that all access-paths to certain JavaScript functionality can be restricted, because the aspect library it relies on was not designed with security in mind. JSand uses the Membrane pattern instead, which was designed to provide complete mediation.

Secure ECMAScript (SES) [20] is a subset of ES5 strict which provides an object-capability language. Unlike Caja, from which it originated, SES runs completely on the client-side without any browser modifications. To the best of our knowledge, JSand is the first fully functional JavaScript integration technique built on SES, capable of handling legacy scripts such as Google Maps and Google Analytics.

7. CONCLUSION

This paper introduced JSand, a server-driven but client-side JavaScript sandboxing framework that does not rely on any browser modifications. We have implemented a prototype of this framework and evaluated it on the most widespread JavaScript scripts around. Although there has been a lot of activity in this research area, we are the first to deliver a solution that provides complete mediation, backwards compatibility and an acceptable performance overhead.

Acknowledgements

This research is partially funded by the Research Fund KU Leuven, the EU-funded FP7 projects NESSoS and WebSand and by the IWT-SBO project SPION. Pieter Agten holds a Ph.D. fellowship of the Research Foundation - Flanders (FWO). With the financial support from the Prevention of and Fight against Crime Programme of the European Union European Commission - Directorate-General Home Affairs.

8. REFERENCES

[1] R. Berjon. W3C HTML5 Working Draft. http://www.w3.org/TR/html5/, September 2012.

[2] BuiltWith. jQuery Usage Statistics. http://trends.builtwith.com/javascript/jQuery.

[3] D. Crockford. ADsafe – making JavaScript safe for advertising. http://adsafe.org/.

[4] T. V. Cutsem and M. S. Miller. On the Design of the ECMAScript Reflection API. Technical Report VUB-SOFT-TR-12-03, Department of Computer Science, Vrije Universiteit Brussel, February 2012.

[5] W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens. FlowFox: a web browser with flexible and precise information flow control. In *Proc. of CCS'12*. ACM, 2012.

[6] P. De Ryck, M. Decat, L. Desmet, F. Piessens, and W. Joosen. Security of web mashups: a survey. In *Proc. of NordSec'10*. Springer, 2011.

[7] D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *Proc of SP'10*, IEEE, pages 109–124, Washington, DC, USA, 2010.

[8] M. Heiderich. Locking the Throne Room - How ES5+ will change XSS and Client Side Security. http://www.slideshare.net/x00mario/locking-the-throneroom-20, November 2011.

[9] Jacaranda. Jacaranda. http://jacaranda.org.

[10] T. Jim, N. Swamy, and M. Hicks. Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. In *Proc. of WWW'07*, pages 601–610, New York, NY, USA, 2007. ACM.

[11] John Resig. Pure JavaScript HTML Parser. http://ejohn.org/blog/pure-javascript-html-parser/.

[12] A. Klein. DOM Based Cross Site Scripting or XSS of the Third Kind. http://www.webappsec.org/projects/articles/071105.shtml, April 2005.

[13] T. Luo and W. Du. Contego: capability-based access control for web browsers. TRUST'11, pages 231–238, Berlin, Heidelberg, 2011. Springer-Verlag.

[14] S. Maffeis, J. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. In *Proc. of SP'10*. IEEE, 2010.

[15] S. Maffeis and A. Taly. Language-based isolation of untrusted Javascript. In *Proc. of CSF'09*, IEEE, 2009.

[16] J. Magazinius, P. Phung, and D. Sands. Safe wrappers and sane policies for self protecting JavaScript. In *Proc. of Nordsec'10*, 2010.

[17] L. Meyerovich and B. Livshits. ConScript: Specifying and enforcing fine-grained security policies for Javascript in the browser. In *Proc. of SP'10*, 2010.

[18] Microsoft Live Labs. Live Labs Websandbox. http://websandbox.org.

[19] Mihai Bazon. UglifyJS. https://github.com/mishoo/UglifyJS/.

[20] M. S. Miller. Secure EcmaScript 5. http://code.google.com/p/es-lab/wiki/SecureEcmaScript.

[21] M. S. Miller. *Robust composition: towards a unified approach to access control and concurrency control*. PhD thesis, Johns Hopkins University, Baltimore, MD, USA, 2006. AAI3245526.

[22] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja - safe active content in sanitized JavaScript. Technical report, Google Inc., June 2008.

[23] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. You are what you include: Large-scale evaluation of remote JavaScript inclusions. In *Proc. of CCS'12*, October 2012.

[24] NoMoreSleep. jquery-geolocation. http://code.google.com/p/jquery-geolocation/.

[25] P. H. Phung and L. Desmet. A two-tier sandbox architecture for untrusted javascript. In *Proc. of JSTools'12*, pages 1–10, New York, NY, 2012. ACM.

[26] P. H. Phung, D. Sands, and A. Chudnov. Lightweight self-protecting JavaScript. ASIACCS '09, pages 47–60, New York, NY, USA, 2009. ACM.

[27] J. G. Politz, S. A. Eliopoulos, A. Guha, and S. Krishnamurthi. ADsafety: type-based verification of JavaScript Sandboxing. In *Proc. of USENIX'11*, SEC'11, pages 12–12, Berkeley, CA, USA, 2011.

[28] Programmable Web. Keeping you up to date with APIs, mashups and the Web as platform. http://www.programmableweb.com/.

[29] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: vulnerability-driven filtering of dynamic HTML. In *Proc. of OSDI'06*, pages 61–74, Berkeley, CA, USA, 2006. USENIX Association.

[30] SANS Institute. SANS: Top Cyber Security Risks. http://www.sans.org/top-cyber-security-risks/, 2009.

[31] S. Stamm, B. Sterne, and G. Markham. Reining in the web with content security policy. In *Proc. of WWW'10*, pages 921–930, New York, NY, 2010. ACM.

[32] M. Ter Louw, K. T. Ganesh, and V. Venkatakrishnan. AdJail: Practical Enforcement of Confidentiality and Integrity Policies on Web Advertisements. In *19th USENIX Security Symposium*, Aug. 2010.

[33] The FaceBook Team. FBJS. http://wiki.developers.facebook.com/index.php/FBJS.

[34] S. Van Acker, P. De Ryck, L. Desmet, F. Piessens, and W. Joosen. WebJail: least-privilege integration of third-party components in web mashups. ACSAC '11, pages 307–316, New York, NY, USA, 2011. ACM.

[35] T. Van Cutsem and M. S. Miller. Proxies: design principles for robust object-oriented intercession APIs. *SIGPLAN Not.*, 45(12):59–72, Oct. 2010.

[36] W3C. Document Object Model (DOM) Technical Reports. http://www.w3.org/DOM/DOMTR.

[37] W3C. W3C Standards and drafts - Cross-Origin Resource Sharing. http://www.w3.org/TR/cors/.

[38] W3C. W3C Standards and drafts - Uniform Messaging Policy, Level One. http://www.w3.org/TR/UMP/.

[39] Yahoo! Developer Network. JavaScript: Use a Web Proxy for Cross-Domain XMLHttpRequest Calls. http://developer.yahoo.com/javascript/howto-proxy.html.

[40] C. Yue and H. Wang. Characterizing Insecure JavaScript Practices on the Web. In *Proc. of WWW'09*, pages 961–961, April 2009.

[41] M. Zalewski. Browser Security Handbook. http://code.google.com/p/browsersec/wiki/Main.

One Year of SSL Internet Measurement

Olivier Levillain*† Arnaud Ébalard* Benjamin Morin* Hervé Debar†

*ANSSI
51 boulebard Latour Maubourg
Paris, France
first.last@ssi.gouv.fr

†Télécom Sud Paris
9 rue Charles Fourier
Evry, France
first.last@telecom-sudparis.eu

ABSTRACT

Over the years, SSL/TLS has become an essential part of internet security. As such, it should offer robust and state-of-the-art security, in particular for HTTPS, its first application. Theoretically, the protocol allows for a trade-off between secure algorithms and decent performance. Yet in practice, servers do not always support the latest version of the protocol, nor do they all enforce strong cryptographic algorithms.

To assess the quality of HTTPS servers in the wild, we enumerated HTTPS servers on the internet in July 2010 and July 2011. We sent several stimuli to the servers to gather detailed information. We then analysed some parameters of the collected data and looked at how they evolved. We also focused on two subsets of TLS hosts within our measure: the trusted hosts (possessing a valid certificate at the time of the probing) and the EV hosts (presenting a trusted, so-called *Extended Validation* certificate). Our contributions rely on this methodology: the stimuli we sent, the criteria we studied and the subsets we focused on.

Moreover, even if EV servers present a somewhat improved certificate quality over the TLS hosts, we show they do not offer overall high quality sessions, which could and should be improved.

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols; D.2.8 [**Software Engineering**]: Metrics

Keywords

SSL/TLS; HTTPS; Internet measure; certificates; X.509

1. INTRODUCTION

SSL (Secure Sockets Layer) is a cryptographic protocol designed by Netscape in 1995 to protect the confidentiality and integrity of HTTP connections. Since 2001, the protocol has been maintained by the IETF (Internet Engineering Task Force) and has been renamed TLS (Transport Layer Security).

SSL/TLS primary objective was to secure online-shopping and banking web sites. With the so-called Web 2.0, its usage has broadened drastically: services provided by Google, Yahoo!, Facebook or Twitter now offer a secure access using TLS. Furthermore, other services like SMTP or IMAP benefit from the security layer; there also exists several VPN (Virtual Private Network) implementations relying on SSL; finally, some Wifi access points use TLS as an authentication protocol (EAP-TLS).

Several flaws have been discovered in TLS, leading to revisions of the standard. Moreover, TLS is subject to various configuration and implementation errors. As TLS usage is so ubiquitous on the internet, it is legitimate to assess its security. Since HTTPS still represents most of the daily TLS usage, we designed our experiments to get a clear view of what browsers face on a daily basis, and whether this view is satisfying or not. We performed several campaigns in 2010 and 2011, to enumerate the HTTPS servers answering on TCP port 443. We used different stimuli to gather precise information about what was effectively supported.

Many SSL/TLS handshake parameters can be considered to assess the quality of a server's answer. Some of them are related to the protocol (version, ciphersuite, extensions) and others can be found in certificate chains (signature algorithms, root certificate, X.509 extensions). We selected various criteria and looked at them through three different subsets of the measures: all the hosts, hosts presenting trusted valid certificates, hosts presenting EV certificates.

Our contribution is therefore threefold:

- using several stimuli to enrich the data collected;
- proposing a variety of criteria to assess TLS quality;
- analysing the data through different subsets.

As such, our work is in line with the suggestions of several researchers in cybersecurity, who advocate that the field would benefit from thorough experiments (e.g., last year's keynote speaker in ACSAC 2011 [1]).

2. STATE OF THE ART

2.1 SSL/TLS: a quick tour

SSL (Secure Sockets Layer) is a protocol originally developed by Netscape in 1995 to secure HTTP connections using a new scheme, `https://`. The first published version was SSLv2 [17], rapidly followed by SSLv3 [16], which fixed

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

Version	Comments
SSLv2	Major structural flaws [29]. Should not be used anymore.
SSLv3	PKCS#1 flaw in early implementations [4]. Interoperability issues with newer versions.
TLSv1.0	Weakness of CBC using implicit IV [23, 12]. Workarounds exist in major software.
TLSv1.1	Minimum recommended version.
TLSv1.2	New ciphersuites (GCM mode, HMAC with SHA2 hash functions).

Table 1: Summary of SSL/TLS versions.

major conceptual flaws. Even if a compatibility mode was described, SSLv2 and SSLv3 use different message formats.

In 2001, the evolution and the maintenance of the protocol were handed to the IETF (Internet Engineering Task Force) which renamed it TLS (Transport Layer Security). TLSv1.0 [9] can be seen as a minor editorial update of SSLv3. TLSv1.1 was published in 2006 [10] and TLSv1.2 in 2008 [11]. Table 1 summarizes the different versions of SSL/TLS. Today, SSLv2 and SSLv3 should not be used anymore, and TLS versions 1.1 and 1.2 should be preferred.

To establish a secure session between a client and a server, SSL/TLS uses handshake messages to negotiate its parameters: the version of the protocol, the cryptographic algorithms and the associated keys. The algorithms are described by so-called *ciphersuites* which define how to:

- authenticate the server[1];
- establish a shared secret used to derive keys;
- encrypt the application data;
- ensure the integrity of the application data.

Figure 1 presents a handshake between a client and a server. First, the client contacts the server over TCP and proposes several versions and ciphersuites; this initial message, `ClientHello`, also contains a nonce. If the server finds an acceptable ciphersuite, it responds with several messages: `ServerHello`, containing the selected version and ciphersuite, the `Certificate` message, containing the chain of certificates for the site contacted, and an empty `ServerHelloDone` message ending the server answer. Then, the client checks the certificates received and sends a `ClientKeyExchange` message, carrying a random value encrypted with the public key of the server[2]. At this point, the client and the server share this secret value, since the server can decrypt the `ClientKeyExchange` message. Finally, the `ChangeCipherSpec` messages activate the negotiated suite and keys, and the `Finished` messages ensure the integrity of the handshake *a posteriori*, as they contain a hash of all the handshake messages previously exchanged. `Finished` messages

[1]Mutual authentication is possible with TLS, but the algorithms used to authenticate the client are negotiated independently in the `CertificateRequest` message. This aspect of TLS is out of the scope of this article.

[2]For the sake of simplicity, the negotiation presented here uses RSA encryption as key exchange algorithm, but other mechanisms exist, like DHE-RSA where an ephemeral Diffie-Hellman is signed by the server with its private key.

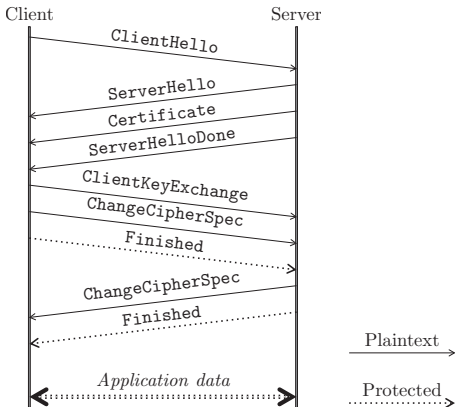


Figure 1: Example of a TLS negotiation.

are the first ones protected with the algorithms and keys negotiated.

At any moment, an `Alert` message can be sent to signal a problem, for example if no ciphersuite is acceptable, or if the client doesn't validate the certificate sent by the server.

The certificates used in TLS follow the X.509 standard [7]. The TLS Public Key Infrastructure is based on several root authorities trusted by default by clients like web browsers.

2.2 Known vulnerabilities

Early SSL/TLS versions had protocol issues. SSLv2 is flawed in numerous ways, the most problematic vulnerability being that an attacker can easily tamper with the negotiation [29]. More recently, Ray devised an attack on the renegotiation feature affecting all TLS versions [24, 25].

SSL/TLS uses many cryptographic primitives, some of which are weak, like DES or MD5 algorithms, and should not be used anymore [15, 28]. Other algorithms were wrongly implemented — Bleichenbacher described in 1998 an attack on PKCS#1 [4] — or incorrectly specified — Rogaway showed in 2002 that an adaptative chosen plaintext attack was possible on CBC, which was patched in TLSv1.1 and later proved to be exploitable in 2011 [23, 12].

It is also possible to encounter symmetric or asymmetric weak keys which lead to the loss of confidentiality in some cases (e.g. RC4 40-bit keys present in *export* ciphersuites), or allow the attacker to control the connection entirely (today, 512-bit RSA keys can easily be factored then used either for a man-in-the-middle attack or for offline decryption).

Since the authentication of the visited sites relies on certificates, processes of generation, validation and revocation thereof are critical.

- Examples of bad random number generators exist: a bug in the Debian version of OpenSSL reduced the effective entropy to only a few dozens of bits from 2006 to 2008 [8]; more recently, Lenstra et al. showed that some network devices did not produce enough entropy and reused prime numbers between two RSA generations, allowing moduli to be factored by a simple gcd algorithm [20].
- X.509 is a standard with many extensions, which have not always been correctly interpreted. For instance,

the `BasicConstraints` extension is used to distinguish server certificate from authority certificates; in 2002, Marlinspike showed that the distinction was not implemented in Webkit nor CryptoAPI [21]. He showed other vulnerabilities on the major SSL/TLS implementations [22].

- Finally, recent incidents affecting certification authorities [6, 30] have proved that the revocation system using CRLs (Certificate Revocation Lists) or OCSP (Online Certificate Status Protocol) did not really function: web browsers had to resort to black lists to limit the consequences of compromised certificates.

The TLS ecosystem is complex and it can be difficult for a client to assess a server's trustworthiness. This is particularly true for web browsers, which face a lot of servers which they have no prior knowledge of. That is why we decided to evaluate what web browsers could encounter out there. Several research teams recently led such campaigns in parallel. These studies are examined in section 8.

3. METHODOLOGY OF THE MEASURES

3.1 Enumerating HTTPS hosts

Gathering data about what a browser faces on a daily basis can be done in several ways:

- enumerating every routable address in the IPv4 space to find open HTTPS ports (TCP/443);
- contacting HTTPS hosts based on a list of DNS (Domain Name System) hostnames;
- collecting real HTTPS traffic from consenting users.

The first method is the most exhaustive, because it tests every IP in the world. However, it leads to contacting many non-HTTPS hosts. Also, it does not take into account the popularity of internet sites (i.e., discriminate sites like `www.google.com` from `randomhost.dyndns.org` or even an unnamed host).

The second option is more restrictive, but better represents user needs, and the proportion of HTTPS servers among the hosts to contact is highly optimised. Besides, this method is compliant with the TLS SNI (Server Name Indication) extension [3], which allows a client to contact different virtual hosts at the same address.

Finally, the last one is completely passive and is really centered on users' habits. In this case it is important to have access to the traffic of many different consenting users to get relevant data that would be comparable to other studies.

We chose the first method to acquire a broad vision of the HTTPS world. This method also allowed us to get consistent answers to multiple stimuli for each given host.

3.2 Description of the campaigns

In July 2010 and July 2011, we launched several campaigns to enumerate HTTPS hosts present in the IPv4 address space. We used different *stimuli* (different `ClientHello`) to grasp the behaviour of the different TLS stacks encountered.

Phase 1: finding the HTTPS hosts

The first task was to find out which hosts were accepting connections on TCP port 443. Using BGP (Border Gateway Protocol) internet routing tables, we reduced the search space from 4 billion IPv4 addresses (2^{32}) to 2 billion routable addresses. Instead of using existing tools such as `nmap` to enumerate open 443 ports, we developped homemade probes to randomize the set of routable addresses globally. For each host, the test consisted simply in a SYN-probe to determine open ports.

To prevent this first phase from being too intrusive, we bounded our upstream rate at 100 kB/s, allowing us to explore the 2 billion addresses in about two weeks.

Phase 2: TLS sessions

Once a host offering a service on port 443 was discovered, we tried to communicate with it using one (or several) TLS `ClientHello`. In this second phase, we used a full TCP handshake followed by several packets, but only with the fraction of servers listening on port 443 (about 1 percent). The second phase could thus be run in parallel with the first one.

To limit the computational impact on servers, we only recorded the first server answer (messages between `ServerHello` and `ServerHelloDone`) before ending the connection. This way, we collected the protocol and ciphersuite chosen by the server, as well as the certificate chain sent.

In the 2010 campaign, we sent only one `ClientHello` message. On the contrary, as we were interested in server behaviour, in the July 2011 campaign, we sent several `ClientHello` messages containing different protocol versions, ciphersuites and TLS extensions.

In addition to our samples, two campaigns were publicly released by the Electronic Frontier Foundation (EFF) in December 2010, allowing us to extend the data to analyse, since they employed a similar methodology to contact the servers and record the answers [13]. Table 2 describes the specificities of the `ClientHello` sent for each dataset. It contains our campaign from July 2010 (NoExt1), our seven campaigns from July 2011 (NoExt2, DHE, FF, EC, SSL2, SSL2+ and TLS12) and also includes both EFF campaigns in italics.

3.3 Issues encountered

Our July 2010 and July 2011 campaigns each took two to three weeks to complete. As explained earlier, this was necessary to avoid link saturation during the host enumeration. However, spanning our measures over several weeks has an impact on the picture of the internet we are seeing. In fact, while probing the different hosts, three factors need to be taken into account:

- the time spent acquiring the data; as the exposure time in photography, it should ideally be as short as possible, to get consistent data;
- the network load induced; sending too many packets can result in some of them getting lost at either end of the connection;
- the use of dynamic IPs in some address blocks; some ISPs change IP addresses every day or so.

Considering the network bandwidth at our disposal and the way IPs were globally randomized, we are confident we did

Id	Date	SSLv2	Max version	Ciphersuites	Extensions
NoExt1	2010/07	no	TLSv1.0	Standard Firefox suites	None
EFF-1	*2010/08*	*yes*	*TLSv1.0*	*SSLv2 + some TLSv1.0 suites*	*None*
EFF-2	*2010/12*	*yes*	*TLSv1.0*	*SSLv2 + some TLSv1.0 suites*	*None*
NoExt2	2011/07	no	TLSv1.0	Standard Firefox suites	None
DHE	2011/07	no	TLSv1.0	DHE suites only	None
FF	2011/07	no	TLSv1.0	Standard Firefox suites	EC, Reneg, Ticket
EC	2011/07	no	TLSv1.0	EC suites only	EC
SSL2	2011/07	yes	SSLv2	SSLv2 suites only	None
SSL2+	2011/07	yes	TLSv1.0	SSLv2 + some TLSv1.0 suites	Reneg
TLS12	2011/07	no	TLSv1.2	mostly TLSv1.2 suites	EC, Reneg, Ticket

Table 2: Different `ClientHello` messages sent during the campaigns. The campaigns in italics were made by the EFF. The SSLv2 column indicates that a SSLv2-compatible `ClientHello` was sent. EC means Elliptic Curves; DHE stands for Diffie-Hellman Ephemeral; finally, Reneg corresponds to the renegotiation extension [25], and Ticket to the Session Ticket one [27].

not overload links during the first phase of the campaigns. We believe that it is as close as we could get to a time-coherent snapshot. Section 6 answers questions about the impact of time on IP address stability.

Even with this in mind, randomization can be insufficient and our SYN packets may be interpreted as an attack, and our IP filtered out. A solution could have been to use several source IPs. Yet, if these addresses were not located in the same neighbourhood, we might have ended up measuring different inconsistent views of the internet, as shown in [18].

Finally, one aspect of gathering such data we did not anticipate was data storage. One easy way is to use one file per active IP, but this rapidly fills up inode/block tables, while the answers to one stimulus only take 20 GB in total. We ended up grouping answers by /8 IP ranges and developping tools to work on such files.

3.4 Global statistics on the campaigns

We first sorted the answers received for each campaign into several categories. Table 3 shows global results for the ten campaigns. It partitions the answers obtained into the following classes.

Non-TLS answers are refined in *empty* and *non-empty* answers (HTTP headers or synctactically invalid TLS messages for example). In fact, it seems common to find non-HTTPS service listening on port 443.

TLS answers can be of three types: *TLS Alerts*, *compatible TLS Handshake messages* or *incompatible TLS Handshake messages* (a `ServerHello` is considered incompatible if it contains a protocol version, a ciphersuite or extensions the client did not propose in its `ClientHello`).

If we compare the EFF-1 and SSL2+ campaigns, which correspond to similar stimuli, we obtain in both cases around 11 million valid TLS answers and 17 thousand TLS alerts, which corroborates the results, but it seems the data from the EFF-1 campaign have been post-processed to eliminate a significant part of the non-TLS results.

On the other hand, it is difficult to compare our results to the other EFF campaign, as the set of IPs probed is significantly smaller. This point is further discussed in section 6.

4. ANALYSIS METHODOLOGY

We first define subsets of the hosts contacted: the TLS hosts, the trusted hosts and the EV hosts. To assess the quality of HTTPS answers, we select several parameters regarding the TLS protocol and the certificate chain.

4.1 Subsets

TLS hosts: hosts that answered with a TLS handshake, compatible or not with the `ClientHello`.

Trusted hosts: servers which presented a server certificate for which we could build a valid chain up to a root certificate present in Firefox[3] and valid at the time of the campaign. Trusted chains not only correspond to RFC-compliant chains, but also to chains containing useless or unordered certificates and even to chains missing links[4]. Accepting the latter servers as trusted hosts conforms to most browsers' behaviour since they cache intermediate CA certificates to allow so-called *path discovery*.

`EV hosts:` EV certificates are a novelty in the internet PKI officially introduced in 2007, which aim at improving the quality of certificates. The EV guidelines [5] describe how the certificates should be issued, the audit procedures needed for the certificate authorities and the cryptographic algorithms that should be banned. An EV certificate must be issued by an EV authority (recognized by the browsers) and contain a certificate policy[5] matching the EV authority. Once a certificate is validated and recognized as EV, the browsers typically use green address bars to indicate to the user that the site is EV-trusted. The EV subset consists of hosts that sent EV chains valid at the time of the campaign. Obviously, EV hosts are a subset of the trusted hosts.

4.2 Criteria studied

Protocol version.

A `ClientHello` message includes two version fields: the external version used for the transport of this particular message (present in the so-called `Record` protocol since SSLv3) and the maximum version supported by the client, v_{max}. The standard indicates that the server should choose the

[3]This particular certificate store is easy to access and representative of many users.

[4]Of course, to build the chain up to a root certificate, the missing links had to be present in another certificate chain or in the root certificate store.

[5]Certificate policies are object identifiers (OID) contained in the `CertificatePolicies` X.509 extension.

Id	IPs with TCP/443	Non-TLS answers		TLS answers					
		Empty	Non-empty	Alerts		Incompat. HS		Compat. HS	
NoExt1	21,342,205	**40.62 %**	**13.20 %**	171,405	**0.80 %**	419	**0.00 %**	9,683,050	**45.37 %**
EFF-1	*15,579,266*	**0.00 %**	**26.79 %**	*16,509*	**0.10 %**	*88*	**0.00 %**	*11,388,447*	**73.10 %**
EFF-2	*7,777,511*	**0.00 %**	**0.91 %**	*591*	**0.00 %**	*48*	**0.00 %**	*7,705,488*	**99.07 %**
NoExt2	26,218,653	**47.28 %**	**9.35 %**	53,535	**0.20 %**	1,018	**0.00 %**	11,313,085	**43.14 %**
DHE	26,218,653	**62.79 %**	**3.19 %**	4,385,635	**16.72 %**	110,606	**0.42 %**	4,421,695	**16.86 %**
FF	26,218,653	**47.36 %**	**9.32 %**	92,239	**0.35 %**	1,007	**0.00 %**	11,262,138	**42.95 %**
EC	26,218,653	**54.26 %**	**9.36 %**	8,696,833	**33.17 %**	142,007	**0.54 %**	695,158	**2.65 %**
SSL2	26,218,653	**70.23 %**	**11.10 %**	354,760	**1.35 %**	3,826	**0.01 %**	4,533,396	**17.29 %**
SSL2+	26,218,653	**47.25 %**	**9.34 %**	17,247	**0.06 %**	127	**0.00 %**	11,361,199	**43.33 %**
TLS12	26,218,653	**54.05 %**	**9.40 %**	918,202	**3.50 %**	108,557	**0.41 %**	8,554,180	**32.62 %**

Table 3: Distribution of the answers collected for each campaign. The percentages are computed over the total number of IPs where we found a 443 open port (second column).

maximum version it supports, up to v_{max}. If no such version exists, it should terminate the handshake with an alert. We also consider that a good server should ban SSLv2.

Ciphersuite.

The range of existing ciphersuites is defined by the IANA, and currently consists of 297 suites[6]. Among those suites, the client generally offers between 10 and 30 suites. In its `ServerHello`, the server will select one.

After eliminating a lot of suites that contain seldom used algorithms (fixed Diffie-Hellman, PSK[7], SRP[8], etc.), we classify the ciphersuites in three groups: weak suites (SSLv2, DES, export suites), acceptable suites (RC4, 3DES, DSS, MD5) and strong ones (the remaining ones). We thus obtain 23 weak suites, 23 acceptable suites and 44 strong suites. Naturally, we would like the servers to select strong suites.

TLS Extensions.

As studied in a recent RFC draft [19], some servers do not support TLS extensions and among them, some implementations simply reject `ClientHello` containing extensions. This is unfortunate since this mechanism is necessary for the extensibility of the protocol:

- security fixes like secure renegotiation [25];
- support for new ciphersuites using elliptic curves [2];
- new features like session tickets [27].

From a security standpoint, when a client proposes the secure renegotiation extension, the server should support it.

Our first analysis focuses on these TLS parameters: the protocol version, the strength of the chosen ciphersuite and support for the renegotiation extension. The results are presented in section 5.

Server behaviour.

In July 2011, as we sent servers multiple `ClientHello`s, we gained unique insight into the behaviour of servers: the versions they support, their reaction to restricted ciphersuite choices, their intolerance to versions and extensions.

[6]To be complete, we must also count the 7 SSLv2 suites and the 4 SSLv3 FIPS suites which are now obsolete or redundant.

[7]Pre-Shared Key.

[8]Secure Remote Password.

Section 6 gives precise statistics about these seven campaigns, by comparing the answers obtained for the different stimuli for each IP in the considered subset (TLS hosts, trusted hosts and EV hosts).

The quality of the certificate chain.

The `Certificate` message contains a list of certificates used to establish the identity of the server. This list should be strictly ordered: the first certificate is the server certificate, and each certificate is signed by the following one. The final root certificate may be omitted, since the client should already know it to trust the certificate chain.

However, some servers do not follow these rules: they forget intermediate certification authorities, they send unordered chains or useless certificates; others even send two certification paths to have a server certificate validated by two root authorities. Such behaviour is generally accepted by common TLS stacks which are laxist and provide features like *path discovery* but may break some implementations[9].

To create the trusted hosts subset, we first need to classify the certificate chains into three groups: empty or incomplete chains, trusted chains, complete but untrusted chains. The two latter classes are further refined:

- *RFC-compliant* chains do contain only the useful certificates in the correct order;
- *self-contained* chains contain useless or unordered certificates, but include all the information needed to build a complete chain;
- *transvalid*[10] chains lack *intermediate* authority to build a complete chain. To reach the root certificate, we thus need to use certificates from other chains.

In addition to this classification, section 7 contains statistics for other parameters: the cryptographic algorithms, the key sizes and the validity period of the chain built.

5. ANALYSIS OF TLS PARAMETERS

5.1 Protocol version

Table 4 shows the distribution of the versions chosen by servers for each campaign and each subset.

For standard `ClientHello` messages (i.e. NoExt1, EFF-1, EFF-2, NoExt2, FF, SSL2+), we observe the same results

[9]One such implementation is the TLS stack present in Java

	TLS		Trusted		EV	
NoExt1	TLS1	96 %	TLS1	99.3 %	TLS1	99.2 %
	SSL3	4 %	SSL3	0.7 %	SSL3	0.8 %
EFF-1	TLS1	95.4 %	TLS1	99.2 %	TLS1	98.4 %
	SSL3	4.5 %	SSL3	0.8 %	SSL3	1.6 %
	SSL2	0.1 %				
EFF-2	TLS1	96 %	TLS1	99.1 %	TLS1	98.5 %
	SSL3	4 %	SSL3	0.9 %	SSL3	1.5 %
NoExt2	TLS1	96 %	TLS1	99.4 %	TLS1	99.4 %
	SSL3	4 %	SSL3	0.6 %	SSL3	0.4 %
DHE⋆	TLS1	97 %	TLS1	99.8 %	TLS1	99.6 %
	SSL3	3 %	SSL3	0.2 %	SSL3	0.4 %
FF	TLS1	96 %	TLS1	99.4 %	TLS1	99.4 %
	SSL3	4 %	SSL3	0.6 %	SSL3	0.4 %
EC⋆	TLS1	84 %	TLS1	100 %	TLS1	100 %
	SSL3	16 %				
SSL2⋆	SSL2	99.9 %	SSL2	100 %	SSL2	100 %
	SSL3	<0.1 %				
	TLS1	<0.1 %				
SSL2+	TLS1	96 %	TLS1	99.2 %	TLS1	99.4 %
	SSL3	4 %	SSL3	0.8 %	SSL3	0.6 %
	SSL2	<0.1 %				
TLS12⋆	TLS1	98.5 %	TLS1	99.6 %	TLS1	99.5 %
	SSL3	1.4 %	SSL3	0.2 %	SSL3	0.2 %
	TLS1.1	0.1 %	TLS1.1	0.2 %	TLS1.1	0.2 %
	TLS1.2	<0.1%	TLS1.2	<0.1%	TLS1.2	<0.1%

Table 4: Distribution of the TLS versions chosen by the servers for each campaign and each subset. Campaigns with a star correspond to specific July 2011 stimuli that produced significantly less answers than NoExt2 or FF.

over time. **TLSv1.0 is the preferred version of the protocol: 95 % of the TLS hosts answer with TLSv1.0 and 5 % use SSLv3.** If we consider the same stimuli but focus on trusted or EV hosts, the proportion becomes more or less 99 % for TLSv1.0 and 1 % for SSLv3. It is worth noting that the results do not change significantly when extensions are sent, or when a SSLv2 compatibility `ClientHello` is used.

Yet, there are still many servers that do not use TLSv1.0 and choose the obsolete SSLv3. Even if the situation is better with trusted and EV hosts, such configurations should be avoided. For the SSL2+ stimulus, this represents 32,000 servers amongst the 4 million trusted servers, and about 1,000 servers amongst the 140,000 EV hosts.

The other stimuli are more difficult to interpret. Indeed, as table 3 shows, the results obtained for DHE, EC, SSL2 and TLS12 stimuli correspond to fewer TLS answers. It shows that some servers refuse to negotiate some protocol versions or some ciphersuites. Among these servers, some will emit an alert, in compliance with the standards. Others will not answer or return inconsistent TLS messages: we call this intolerance to particular versions/suites. Section 6 discusses this matter further.

5.2 Ciphersuites

Table 5 shows the distribution of ciphersuites chosen by servers for each campaign and each subset. The ciphersuites are grouped into categories: S (strong), A (acceptable), W (weak)[11] and N represents suites that were *not* proposed by the client.

	TLS		Trusted		EV	
NoExt1	S	65 %	S	59 %	S	59 %
	A	35 %	A	41 %	A	41 %
EFF-1	S	64 %	S	62 %	S	54 %
	A	36 %	A	38 %	A	46 %
EFF-2	S	64 %	S	61 %	S	56 %
	A	36 %	A	39 %	A	44 %
NoExt2	S	73 %	S	68 %	S	80 %
	A	27 %	A	32 %	A	20 %
DHE⋆	S	95 %	S	98.6 %	S	98.9 %
	A	2.5 %	A	1.4 %	A	1.1 %
	N	2.5 %				
FF	S	73 %	S	68 %	S	80 %
	A	27 %	A	32 %	A	20 %
EC⋆	S	83 %	S	99.2 %	S	98.9 %
	N	17 %	N	0.8 %	N	1.1 %
SSL2⋆	W	99.9 %	W	100 %	W	100 %
	N	0.1 %				
SSL2+	S	71 %	S	67 %	S	80 %
	A	29 %	A	33 %	A	20 %
	W	0.1 %				
TLS12⋆	S	98.8 %	S	100 %	S	100 %
	N	1.2 %				

Table 5: Distribution of the ciphersuites chosen by the servers for each campaign and each subset. Percentages below 0.1 % are ignored.

For the first three campaigns (NoExt1, EFF-1 and EFF-2), around 65 % of the answers contain strong suites. For trusted or EV subsets, this proportion is smaller. This discrepancy is explained by the fact that the A category contains the popular `TLS_RSA_WITH_RC4_128_MD5` suite. While we consider this suite only acceptable[12], it is commonly considered the most efficient one. For the NoExt2, FF and SSL2+ campaigns from July 2011, proportions are roughly the same: 72 % for TLS hosts, 68 % for trusted hosts and 80 % for EV hosts, which indicates a minor improvement in the suite choices, especially for EV hosts.

We did not expect servers to answer with ciphersuites *not* proposed in `ClientHello`, as it is not compliant with the specifications. This phenomenon is significant in the DHE and EC campaigns, when the servers faced a limited choice. We also witness this behaviour with TLS12 stimulus, essentially because we chose not to propose the popular `RC4_MD5` ciphersuite[13]. This is a manifestation of server intolerance to DHE/EC/TLSv1.2.

Another way to look at the ciphersuites chosen by servers is to compute the proportion of them that provide Perfect Forward Secrecy (PFS). Such suites do not allow the decryption of past sessions if the server private key is compromised[14]. Results are presented in table 6. We did not

ClassPath.

[10]The term *transvalid* was first used by the EFF to describe such chains.

[11]Such suites were only proposed in the SSL2 and SSL2+ `ClientHello`.

[12]On one hand, statistical vulnerabilities on RC4 limit the quantity of data that should be protected without refreshing the keys. On the other hand, it is strongly discouraged to use MD5 nowadays.

[13]However, other classical non-DHE non-EC suites were proposed in TLS12 `ClientHello`.

[14]With TLS ciphersuites, this property is obtained with

		TLS	Trusted	EV
NoExt1	2010-07	41 %	44 %	20 %
EFF-1	2010-08	43 %	46 %	19 %
EFF-2	2010-12	41 %	42 %	19 %
NoExt2/FF SSL2+	2011-07	37 %	39 %	11 %
TLS12	2011-07	0 %	0 %	0 %

Table 6: Proportion of the ciphersuites chosen by the servers that provide Perfect Forward Secrecy. The percentage is computed over the total number of compatible answers.

include the DHE and EC campaigns (since all the proposed suites offers PFS) nor the SL2 stimulus (since none of the suites proposed offers PFS).

If we look at the first four lines, corresponding to four different dates, around 40 % of TLS hosts chose PFS suites. The corresponding trusted hosts figures are a little higher, whereas the proportions drops with EV hosts, especially in July 2011 where only 11 % of the servers chose PFS.

The result for the TLS12 stimulus can be explained by the fact that the PFS suites proposed in this case were all compatible with TLSv1.2 only, leaving no choice to non-TLSv1.2 implementations but not to offer PFS.

For a standard stimulus, about two thirds of TLS hosts choose strong ciphersuites. Recently, the proportion seems to have grown to 80 % for EV hosts. However, only 40 % of TLS hosts, and less than 20 % of EV hosts choose a suite offering the Perfect Forward Secrecy.

5.3 Secure renegotiation

As discussed in section 4.2, we expect servers to implement RFC 5746 for secure TLS renegotiations. However, only three out of the ten stimuli studied proposed this extension (FF, SSL2+[15] and TLS12), all sent in July 2011. As a result, we can not assess the evolution in time of this parameter.

If we now consider only those three stimuli proposing the extension, we have the same results: 53 % of the TLS hosts accept RFC 5746 extension, which becomes 65 % when we focus on trusted hosts. For EV hosts, the proportion is even better: 80 %. However, 20 % of EV hosts (around 28,000 servers) still do not support secure renegotiation as of July 2011.

It is important to notice that servers are only vulnerable if they do not support the extension *and* if they accept to renegotiate. As we did not pursue the connection, nor tried to renegotiate, we are not able to reliably tell how many servers are indeed vulnerable. Yet, from the client's perspective, the only way to be sure that the server is not vulnerable is if it supports the secure renegotiation extension.

As of July 2011, one third of the trusted hosts and 20 % of EV hosts do not support secure renegotiation.

ephemeral Diffie-Hellman key exchanges, as opposed to the RSA encrypted key exchange.

[15]The SSL2+ `ClientHello` did not propose the extension *per se*, but rather used a dedicated ciphersuite to signal the support of secure renegotiation.

6. ANALYSIS OF SERVER BEHAVIOUR

Using the collected data, we tried to compare the answer types for a given IP in different campaigns. However, we found out that such a comparison was irrelevant when we compared data collected at different times.

6.1 When comparison really makes sense

We compared the list of IPs from the TLS subset between our measure of July 2010 and the EFF measure of August 2010: 7.5 million IPs are present in both sets but 2 million were only seen by our trace, and nearly 4 million were only present in the EFF trace. So a vast proportion of TLS hosts do not have a stable IP over a month or a year.

This fact is even more visible when comparing our measures in 2010 and 2011 (using the NoExt2 measure): 5.5 million IPs correspond to TLS hosts in both cases, but about 4 million (resp. 6 million) IPs are present only in the 2010 (resp. 2011) trace. It is thus clear we can only do per-IP comparisons between measures that were made at the same time.

This basic comparison sheds light on the surprising figures obtained in EFF-2 campaign: between the EFF experiments conducted in August and December, 7.5 million IPs represent TLS hosts both times, about 4 million have disappeared in EFF-2 whereas only 60,000 IPs are new in EFF-2. This can be explained by the fact that the EFF did not launch the first phase (enumerating IP with TCP/443 open) again in December and reused the list of IPs from August 2010.

6.2 Error margin for July 2011 campaigns

We now focus on the seven measures of July 2011 that were conducted simultaneously[16] on the same address pool to understand server answers against different stimuli. We need to check that the servers we contacted were the same during all the communications. Let's first compare the IPs corresponding to a TLS answer between two similar campaigns: NoExt2 and FF. Over 99.6 % of IPs corresponding to a TLS host in one measure also do in the other. The correlation is even better if we focus on trusted or EV hosts.

To confirm that, we also compare the server certificates returned by the servers. Considering the set of IPs that answered at least once with a server certificate, we count for each IP the number of different server certificates received over the seven communications. More than 99.6 % of them presented the same server certificate each time they answered with a valid `ServerHello`. If we compute the same statistics using the list of IPs presenting at least once a trusted certificate (resp. an EV certificate), the error margin is even better, since 99.9 % served only one certificate. The hosts that do not consistently send the same server certificate (0.4 %) define our error margin.

6.3 Understanding hosts with multiple stimuli

We can now refine the statistics about DHE/EC/TLSv1.2 intolerance. Let us focus on the servers that correctly answered with a compatible `ServerHello` to the NoExt2, FF and SSL2+ stimuli. We thus obtain a list of apparently valid IPs for each subset: 11.2 million TLS hosts, 4.0 million trusted hosts and 141,972 EV hosts. Tables 7, 8, 9 present the answers of this sample groups to the DHE, EC

[16]For a given IP, all `ClientHello` messages were sent within a 10 minute-timeframe.

and TLS12 stimuli. The answers are partitioned into the categories defined in section 3.4.

	TLS	Trusted	EV
Compatible Handshake	39 %	42 %	13 %
Alert	38 %	28 %	71 %
Intolerant servers	**23 %**	**30 %**	**16 %**
Non-TLS answer	22 %	30 %	16 %
Incompatible Handshake	1 %	0 %	0 %

Table 7: Answers to the DHE stimulus.

	TLS	Trusted	EV
Compatible Handshake	6 %	10 %	5 %
Alert	76 %	68 %	82 %
Intolerant servers	**18 %**	**22 %**	**13 %**
Non-TLS answer	17 %	22 %	13 %
Incompatible Handshake	1 %	0 %	0 %

Table 8: Answers to the EC stimulus.

	TLS	Trusted	EV
Compatible Handshake	76 %	74 %	86 %
Alert	7 %	5 %	2 %
Intolerant servers	**17 %**	**21 %**	**12 %**
Non-TLS answer	16 %	21 %	12 %
Incompatible Handshake	1 %	0 %	0 %

Table 9: Answers to the TLS12 stimulus.

The Alert line shows the proportion of servers that refuse to negotiate and assert this choice. This should happen if none of the proposed ciphersuites is acceptable. Theoretically, this should not happen with a protocol version, since the servers of the sample groups accepted TLSv1.0 and could have answered with this version. Yet, our TLS12 stimulus did not contain all the suites of the NoExt2/FF/SSL2+ `ClientHello`s so it is legitimate for a server to accept the latter stimuli but send an Alert to the TLS12 message.

The last two lines represent servers that do not respond correctly to the stimulus. As they correctly answered three other stimuli, we would have expected an Alert message to signal the negotiation failure. We call such servers DHE-, EC- or TLSv1.2-intolerant, and their behaviour does not conform to the standards.

For each case (DHE, EC, TLSv1.2), the proportion of intolerant servers is very important: about 20 % globally, more than 12 % for EV servers.

Another disappointing fact is the very low proportions of servers supporting DHE and EC suites, especially for EV hosts (13% for DHE, 5 % for EC).

Finally, let's consider the proportion of the sample groups answering correctly to a SSLv2 `ServerHello` when the stimulus is a pure SSLv2 `ClientHello` (SSL2 stimulus). Table 10 shows the answers received. We did not expect TLS servers to behave correctly, since SSLv2 uses different messages and is now deprecated. In fact, we would have expected fewer servers to accept negotiating a SSLv2 session.

Many TLS servers are still fully compatible with SSLv2, whereas they should not negotiate the obsolete version of the protocol.

	TLS	Trusted	EV
Compatible Handshake	**40 %**	**27 %**	**8 %**
Alert	2 %	2 %	1 %
Non-TLS answer	58 %	71 %	91 %
Incompatible Handshake	0 %	0 %	0 %

Table 10: Answers to the SSL2 stimulus.

7. ANALYSIS OF CHAIN QUALITY

In this section, we only consider four campaigns (NoExt1, EFF-1, EFF-2 and NoExt2) which correspond to different dates (July 2010, August 2010, December 2010 and July 2011). The results of this section were similar for the three standard July 2011 stimuli (NoExt2, FF, SSL2+). For the four campaigns, trusted hosts represent around 35 % of the TLS hosts (about 4 million) and EV servers represent 1 % of the TLS hosts (100 to 140,000). 10.2 million unique certificates were analysed, that were gathered from 10.9 million unique certificate chains.

The certificate chains we study are the chains built by our verification program, i.e. the best certificate chain we could build from the certificates sent and all the certificates gathered. We prefer trusted chains over non-trusted chains, and chose RFC-compliant chains (R) over complete but unordered chains (C) over transvalid chains (T). The partition of certificate chains built along these latter categories are given in table 11. It is interesting to notice that EV hosts often present unordered or even transvalid chains[17], which leads to incompatibilities with some TLS stacks[18]. Generally, servers send 2 or 3 certificates and the certificate chains we build also contain the same number of certificates. However, some servers send more certificates. The maximum we saw was 150 in EFF-2 and corresponded to a trusted server.

More than 40 % TLS hosts do not send RFC-compliant chains. The values for EV hosts are even worse (around 85 %). These observations are stable from 2010 to 2011.

	2010-07	2010-08	2010-12	2011-07
TLS	R : 60 % C : 9 % T : 4 % I : 27 %	R : 61 % C : 8 % T : 3 % I : 28 %	R : 59 % C : 10 % T : 6 % I : 25 %	R : 54 % C : 10 % T : 6 % I : 30 %
Trusted	R : 69 % C : 21 % T : 10 %	R : 71 % C : 19 % T : 10 %	R : 67 % C : 21 % T : 12 %	R : 62 % C : 24 % T : 14 %
EV	R : 11 % C : 78 % T : 11 %	R : 13 % C : 76 % T : 11 %	R : 16 % C : 74 % T : 10 %	R : 12 % C : 83 % T : 5 %

Table 11: Partition of the certificate chains built in (R)FC-compliant, (C)omplete although not RFC-compliant and (T)ransvalid chains. (I)ncomplete chains are chains we could not build completely.

RSA is the main algorithm used in the certificates sent: the proportion of certificate chains containing only RSA keys

[17] As mentioned earlier, transvalid chains are chain missing *intermediate* certificate authorities, the root certificate being optional in the RFC.

[18] For example, the Java TLS implementation can not validate such chains.

is higher than 99 % for TLS hosts, and reaches 100 % for trusted and EV hosts. We thus would like to assess the cryptographic robustness of such RSA certificate chains. The criterium to measure RSA key robustness is the minimum RSA key length found in the certificate chain. The statistics for this parameter are given in table 12. It appears that mean RSA key lengths are increasing with time. The 84 % 1024-bit and 13 % 2048-bit chains measured in 2010 turn into 52 % and 48 % respectively. The shift is even better for Trusted hosts, since we have 86 % of 1024 bit chains and 13 % of 2048 bit chains in July 2010 that have become 52 % and 48 % in July 2011. In 2011, all the EV servers present 2048 bit robust chains, which can be explained by the EV guidelines [5] that specify this as a minimum key size for EV certificates from December 2010.

	2010-07	2010-08	2010-12	2011-07
TLS	1147	1135	1197	1303
Trusted	1149	1133	1220	1514
EV	1950	1897	2042	2048

Table 12: Mean RSA robustness of the chains.

This table does not include extreme values: few servers present huge RSA keys (up to 16384 bit long) or very short keys (512 or 768 bits). Such weak RSA keys represent 3 % of TLS hosts in the first two campaigns and less than 2 % for the last two. What is more serious is that 512 and 768 bit certificates are present in 2 % of trusted hosts in July 2010. Fortunately, this number has dropped to less than 0.1 % (less than 2,500 servers) in July 2011.

Finally, the last parameter we study is the validity period of the chain (i.e. the intersection of the validity periods of the certificates in the chain). The mean values are represented in table 13. As expected, trusted and EV validity periods are reasonable (mostly one or two years). However TLS hosts do contain anomalies (chains that are never valid, or valid until the year 9999), that are hopefully skimmed by the trusted filter. Another trend we observe is that the typical validity of EV certs has moved from 365 days to 730 days between July 2010 and December 2011.

	2010-07	2010-08	2010-12	2011-07
TLS	2561	5020	2328	2659
Trusted	701	728	728	744
EV	551	555	612	652

Table 13: Mean validity period (in days).

The chain validity period and the key robustness are well understood parameters that have improved over time. They have reached acceptable values in the EV subset: at least 2048 bit RSA keys, 1- or 2-year validity. This is fortunate, as EV was designed specifically to take these parameters into account.

8. RELATED WORK

Several projects aiming at understanding the SSL landscape have performed similar measures on the internet. We discuss three of them: two which were presented at security conferences (BlackHat, DefCon and CCC) and one covered in an academic paper.

As explained in section 3, the EFF performed two similar campaigns. They presented their results about the certificates gathered in 2010 [13, 14]. Even if the global methodology resembles ours (enumerating TCP/443 open ports on the IPv4 space), they used standard tools to find TLS hosts (`nmap`) whereas we developped specific tools to fully randomize the IPv4 space. Besides, the set of IPs initally probed was not exactly the same: they used a restricted list of /8 prefixes following the IANA information. Additionally, in their second measure (EFF-2), they did not enumerate the TCP/443 hosts again and used the August 2010 list instead. Another main difference in our approaches is the stimulus: the EFF sent a SSLv2 `ClientHello` whereas we tried to obtain more information by sending different stimuli. Contrary to this study, our work does not focus exclusively on the certificate chains sent by trusted hosts; but we broaden the criteria to assess the quality of servers' answers and to show trends by focusing on different subsets of hosts.

In 2010, Ivan Ristic from Qualys SSL Labs presented another SSL survey focusing this time on a DNS enumeration of HTTPS hosts [26]. He established several connections with each of the servers tested. His goal was to assess the quality of TLS answers from servers reachable via a DNS hostname. His results concerning protocol support match ours: SSLv2 is still widely supported, at least with a compatible `ClientHello`, while TLSv1.1 and TLSv1.2 are virtually inexistant. Our results also concur with his findings about the quality of the `Certificate` message sent by servers that are not strictly RFC-compliant. Like SSL Labs, we work on more criteria than just the certificates, but the difference in host enumeration makes our work complement theirs. In addition, focusing on the EV hosts allows us to present additionnal results. Since April 2012, SSL Labs have launched SSL Pulse, a dashboard providing daily measures for 200,000 HTTPS sites, which gives a partial but interesting insight on trends in SSL deployements.

Finally, in 2011, Ralph Holtz et al. from the University of München gathered and studied different data sets: active probing of popular sites, passive monitoring on a 10Gb link and the EFF campaigns [18]. They provide a thorough analysis of the certificates received for each of the campaigns studied. One of their results is to compare the certificates received by clients from different source addresses and to spot suspicious certificates from these sets. Using real world traffic is a very interesting source of information and once again, it is complementary to full IPv4 host enumerations.

9. CONCLUSION

From July 2010 to July 2011, we gathered data from full IPv4 HTTPS host enumerations, evaluated the quality of TLS answers and described trends in time.

We found that some well studied parameters, like RSA key sizes, are improving, but most of the criteria we analysed are not well taken into account, even if some parameters have improved in a year. For example, a lot of servers are still intolerant to some ciphersuites or to recent TLS versions. The quality of the certificate chains sent by servers is also not acceptable, since many HTTPS hosts send `Certificate` messages that do not comply to the standard, which makes some TLS stacks fail.

There is a pressing need for a quality label representing the overall quality of TLS sessions (server configuration, implementation and cryptographic parameters). The only wide-

spread existing label is Extended Validation, which is visually recognisable in web browsers. However, EV only deals with the format of the certificate issued to servers, and does not take into account the other parameters. In fact, global TLS statistics were even better than EV statistics for some parameters. One way of improving the SSL landscape would thus be to create a new label or to extend EV constraints to cover all the criteria relevant to security: support for recent TLS versions and for most secure ciphersuites, preference for PFS suites, strict RFC-compliance, support of known security extensions.

Recent initiatives like the EFF Decentralized SSL Observatory (passive monitoring through a browser plugin) or SSL Labs' SSL Pulse should help monitoring parts of the SSL landscape, as the EFF calls it. It may also be useful to browse the full IPv4 space again to compare global views of the SSLiverse over time. Indeed, using only DNS scans or passive monitoring does not allow for really comparable statistics.

Further work could also include new stimuli to refine the data obtained. We could study other parameters like the certificate revocation methods. To improve our notion of trust, it would be useful to take other certificate trust stores into account (e.g. Internet Explorer, Opera). Finally, what we noticed was that many servers did not behave like common known TLS stacks, so it would be interesting to investigate their answers to try and fingerprint the stacks encountered.

Acknowledgment

The work in this paper has been partially sponsored by the EC 7th Framework Programme as part of the ICT VisSense project (grant no. 257497). The authors would like to thank the Applied and Fundamental Research Division of the French Network and Information Security Agency (ANSSI) for their comments and suggestions.

10. REFERENCES

[1] T. Benzel. The science of cyber security experimentation: the DETER project. In Robert H'obbes' Zakon, John P. McDermott, and Michael E. Locasto, editors, *ACSAC*, pages 137–148. ACM, 2011.

[2] S. Blake-Wilson, N. Bolyard, V. Gupta, C. Hawk, and B. Moeller. Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS). RFC 4492 (Informational), May 2006.

[3] S. Blake-Wilson, M. Nystrom, D. Hopwood, J. Mikkelsen, and T. Wright. Transport Layer Security (TLS) Extensions. RFC 3546 (Proposed Standard), June 2003.

[4] D. Bleichenbacher. Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1. In *CRYPTO*, 1998.

[5] CA/Browser Forum. EV SSL Certificate Guidelines version 1.3, 2010.

[6] Comodo. Report of Incident - Comodo detected and thwarted an intrusion on 26-MAR-2011, 2011.

[7] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280 (Proposed Standard), May 2008.

[8] Debian. DSA-1571-1 openssl – predictable random number generator, 2008.

[9] T. Dierks and C. Allen. The TLS Protocol Version 1.0. RFC 2246 (Proposed Standard), January 1999.

[10] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346 (Proposed Standard), April 2006.

[11] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008.

[12] T. Duong and J. Rizzo. BEAST: Surprising crypto attack against HTTPS, 2011.

[13] P. Eckersley and J. Burns. An Observatory for the SSLiverse, Talk at Defcon 18, 2010.

[14] P. Eckersley and J. Burns. Is the SSLiverse a safe place?, Talk at 27C3, 2010.

[15] Electronic Frontier Foundation. *Cracking DES. Secrets of Encryption Research, Wiretap Politics & Chip Design*. O'Reilly, 1998.

[16] A. Freier, P. Karlton, and P. Kocher. The SSL Protocol Version 3.0, 1996.

[17] K. Hickman. The SSL Protocol, 1994-1995.

[18] R. Holz, L. Braun, N. Kammenhuber, and G. Carle. The SSL landscape: a thorough analysis of the X.509 PKI using active and passive measurements. In *IMC'11*, 2011.

[19] A. Langley. Unfortunate current practices for HTTP over TLS. Internet Draft, 2011.

[20] A. Lenstra, J. Hughes, M. Augier, J. Bos, T. Kleinjung, and C. Wachter. Ron was wrong, Whit is right. Cryptology ePrint Archive, Report 2012/064, 2012.

[21] M. Marlinspike. Internet Explorer SSL Vulnerability, 2002.

[22] M. Marlinspike. More Tricks For Defeating SSL In Practice, 2009.

[23] B. Moeller. Security of CBC Ciphersuites in SSL/TLS: Problems and Countermeasures, 2002-2004.

[24] M. Ray. Authentication gap in TLS renegotiation, 2009.

[25] E. Rescorla, M. Ray, S. Dispensa, and N. Oskov. Transport Layer Security (TLS) Renegotiation Indication Extension. RFC 5746 (Proposed Standard), February 2010.

[26] I. Ristic. Internet SSL Survey, Talk at BlackHat 2010, 2010.

[27] J. Salowey, H. Zhou, P. Eronen, and H. Tschofenig. Transport Layer Security (TLS) Session Resumption without Server-Side State. RFC 4507 (Proposed Standard), May 2006.

[28] M. Stevens, A. Sotirov, A. Lenstra J. Appelbaum, D. Molnar, D.A. Osvik, and B.D. Weger. Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate. In *Crypto 2009, LNCS 5677*, pages 55–69, 2009.

[29] S. Turner and T. Polk. Prohibiting Secure Sockets Layer (SSL) Version 2.0. RFC 6176 (Proposed Standard), March 2011.

[30] Vasco. DigiNotar reports security incident, 2011.

Dissecting Ghost Clicks: Ad Fraud Via Misdirected Human Clicks

Sumayah A. Alrwais*
Indiana University
Bloomington, U.S.A.
salrwais@cs.indiana.edu

Christopher W. Dunn
Indiana University
Bloomington, U.S.A.
chrdunn@cs.indiana.edu

Minaxi Gupta
Indiana University
Bloomington, U.S.A.
minaxi@cs.indiana.edu

Alexandre Gerber
AT&T Labs-Research, U.S.A.
alexgerber67@gmail.com

Oliver Spatscheck
AT&T Labs-Research, U.S.A.
spatsch@research.att.com

Eric Osterweil
Verisign Labs, U.S.A.
eosterweil@verisign.com

ABSTRACT

FBI's *Operation Ghost Click*, the largest cybercriminal takedown in history, recently took down an ad fraud infrastructure that affected 4 million users and made its owners 14 million USD over a period of four years. The attackers hijacked clicks and ad impressions on victim machines infected by a DNS changer malware to earn ad revenue fraudulently. We experimented with the attack infrastructure when it was in operation and present a detailed account of the attackers' modus operandi. We also study the impact of this attack on real-world users and find that 37 subscriber lines were impacted in our data set. Also, 20 ad networks and 257 legitimate Web content publishers lost ad revenue while the attackers earned revenue convincing a dozen other ad networks that their ads were served on websites with real visitors. Our work expands the understanding of modalities of ad fraud and could help guide appropriate defense strategies.

1 Introduction

Online advertising is a fast growing multi-billion dollar industry. At its core are content *publishers*, such as websites that provide news and search functionality, and *advertisers* which are connected to each other by *advertising networks*. The advertisers pay the advertising networks for displaying their advertisements (or ads, as they are referred to in common parlance). The ad networks act as brokers, finding suitable publishers for ads and then splitting the revenue with them. Common revenue models include: cost per mille (CPM), where advertisers are charged per thousand impressions; cost per click (CPC), where advertisers are charged per click; and cost per action (CPA), where advertisers are charged per action, such as an online sale.

As revenues generated by online advertising have increased, so has the fraudulent activity. Examples of fraudulent activity include publishers generating unwarranted ad revenue through the use of human clickers or botnets. Publishers are also known to attract higher value ads by manipulating keywords on their websites. Further, ill-intended publishers sometimes stack ad impressions on top of each other to collect revenue hurting the advertisers of these ads. On the other hand, advertisers are also known to serve malicious ads and to also hurt their rivals by exhausting their advertising budget [15, 11].

Owing to the seriousness of the problem, a few recent works have examined how ad fraud is perpetrated. Miller et al. investigated a botnet that was used to commit ad fraud and then used that understanding to study how ad networks can identify and combat ad fraud [35]. Complementary work by Stone-Gross et al. studied the operation of two malware families, *Fiesta* and *7cy*, whose bots are known to commit ad fraud by clicking on ads [31]. A common theme in these works are ensembles of *clickbots*, botnets that specialize in committing ad fraud. Zhang et al. recently conducted a study that verified the role of human clickers in ad fraud [42]. Our work expands the landscape of ad fraud beyond these works and proves that ad fraud needs to be understood better in order for the defenses to be adequate.

Specifically, in this paper, we present the dissection of an ad fraud scheme whose infrastructure was recently taken down by the FBI under *Operation Ghost Click*. It was the largest cybercriminal takedown in history [27, 34, 36] that made its attackers 14 million USD over a course of four years. The attackers did not use any clickbots. Instead, real human clicks were misdirected to generate ad revenue fraudulently. The key element in this scheme was a DNS changer malware that changed the DNS resolver setting on victim machines to attacker-controlled resolvers in Eastern Europe. This simple change allowed the attackers to hijack ad impressions and clicks and earn CPM and CPC revenue, all the while fooling tens of ad networks that the revenue was earned on publisher websites owned by the attackers. A subtle distinction of this ad fraud scheme with respect to clickbots was that while clickbots only hurt advertisers, this scheme earned the attackers revenue at the cost of hurting many legitimate publishers, ad networks and advertisers. The key contributions of our work are the following:

- In situ experimentation: We experimented with the attack infrastructure when it was alive and in-action. This allowed us to scrutinize the attackers' modus operandi in depth. We present a detailed account of the operation.

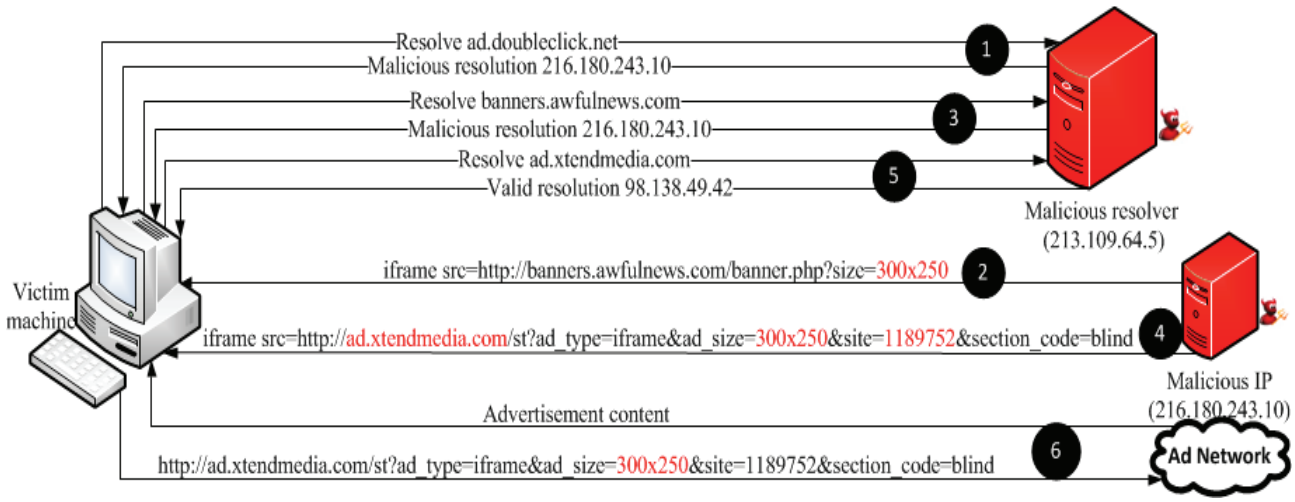
- Mapping the attack infrastructure: We develop a methodology to map the attack infrastructure and find that a total of 1039 malicious DNS resolvers were misdirecting

*Funded by the College of Computer and Information Sciences, King Saud University, Riyadh, Saudi Arabia.
salrwais@ksu.edu.sa

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

```
http://ad.doubleclick.net/adi/ebay.ebayus.homepage.cchp/cchp;sz=300x250;ord=1305150844507;u=i_
9083996730633540328|m_172562;dcopt=ist;tile=1;um=0;us=13;eb_trk=172562;pr=20;xp=20;np=20;uz=;
```

(a) An ad URL on eBay website



(b) Attack chain leading to ad replacement

Figure 1: Steps in the ad replacement attack vector

victims of the DNS-changer malware and the attackers used 30 different IP addresses to commit fraud. These IP addresses were registered with 6 different ISPs. Overall, the attack infrastructure was well provisioned, with attention paid to fault tolerance.

- Gauging attack impact: Using HTTP traces of 17K DSL subscribers of a Tier-1 ISP for a day, we estimate the impact of this fraud. We find that 20 ad networks and 257 legitimate content publishers lost revenue. The attackers stole this revenue by pretending to be a publisher of 423 websites with at least a dozen different ad networks they defrauded. Our estimates of affected users come close to the numbers quoted by the FBI.

-Mitigation: Our work highlights that scrutinizing open recursive resolvers, a characteristic shared by the malicious resolvers used in the attack, could be a fruitful dimension in the fight against ad fraud.

2 Ad Fraud Scheme

The attackers made use of two attack vectors to earn ad revenue fraudulently. In the first, they earned cost per mille (CPM) revenue by replacing displayed ads. In the second, they earned cost per click (CPC) revenue by hijacking clicks. Both the attack vectors worked on victim machines infected by a DNS changer malware [37, 7, 19], which is a malware that utilizes social engineering to download and install a certain codec to play a video. Upon installation, it changes the machine's local DNS resolver to a malicious one. Different variants of DNS changers have surfaced, such as ZCodec [40]. Another way of altering the DNS settings is by using a rogue DHCP which sends the IP address of a rogue DNS server along with the allocated IP to any machine that requests to connect to the network [38, 21]. Yet another variant of the DNS changer attacks routers to change their resolver settings [30]. The latest version of this malware that appeared in 2011 is the Google redirect malware [32] which was reported to hijack Google's search engine results.

To carry out these attacks, the attackers' registered many websites they owned as *publisher* websites with various ad networks. We refer to the attacker-controlled websites as *front-end websites* subsequently in this paper. They also controlled a network of malicious DNS resolvers in Eastern Europe that victim machines contacted instead of legitimate resolvers, say ones provided by their ISPs. The malicious DNS resolvers selectively mis-resolved hosts that serve ads or Javascripts, enabling the two attack vectors. Next, we explain each attack vector through a real-world example.

2.1 Ad Replacement Attack

Consider a victim machine visiting a web page, `ebay.com`. The visit generates an HTTP request for an ad from DoubleClick using the URL shown in Figure 1(a). The ad host in this URL is `ad.doubleclick.net`. Figure 1(b) shows the steps that ensue:

Step-1: The malicious resolver mis-resolves the ad host name, `ad.doubleclick.net`, to `216.180.243.10`.

Step-2: The malicious IP serves an iFrame to the victim. The iFrame directs the victim to a front-end website to serve an ad of the same size as the original ad.

Step-3: The front-end website, `banners.awefulnews.com`, is also mis-resolved to the malicious IP address `216.180.243.10`. The mis-resolution ensures that anyone scrutinizing the front-end website, say an ad network, can be shown legitimate-looking content through a correct resolution.

Step-4: The mis-resolved front-end website serves an iFrame. The iFrame loads an ad URL which belongs to a different ad network, `XtendMedia`. The attackers have an account with this ad network and have convinced the ad network that it will display its ads on the front-end website.

Step-5: The new ad network's host name, `ad.xtendmedia.com`, is resolved correctly.

Step-6: The new ad network serves an ad to the victim which the malicious resolver resolves correctly. The publisher ID on the displayed ad URL is one of the attackers' IDs with that ad network.

At the end of the above series of steps, the attackers collected CPM revenue for displaying an ad from `XtendMedia` on one of their front-end websites. The original ad net-

#	Method	Status	URL	IP	Notes
1	GET	200	http://www.google.com/s?YYYY&**q=download antivirus free**& YYYY	74.125.159.106	Legitimate host names correctly resolved
2	GET	200	http://www.google.com/url?sa=t&YYYY&**url=http://free.avg.com**& YYYY&**q=download antivirus free**& YYYY	74.125.159.107	
3	GET	200	http://free.avg.com/us-en/homepage	63.236.253.27	
4	GET	200	http://www.google-analytics.com/ga.js	205.234.201.229	Legitimate host name mis-resolved
5	GET	200	http://search2.google.com/**123.php**?ref=http://www.google.com/url?sa=t&YYYY&**url=http://free.avg.com**&YYYY&**q=download antivirus free**&YYYY	67.210.14.53	Non-existent host names
6	GET	200	http://search3.google.com/?_kwd=downloadantivirusfree& lnk=http://free.avg.com/us-en/homepage&t=g		
7	GET	200	http://bulletindaily.com/?65287254d575354084f584e49		Front-end website mis-resolved
8	GET	200	http://65.60.9.238/click.php?c=eb951aa106822a13341f45005500	65.60.9.238	Form click IP
9	POST	302	http://www.accurately-locate.com/jump2/? affiliate=6990&subid=191&terms=downloadantivirusfree	67.29.139.153	Fake search engine correctly resolved
10	GET	302	http://r.looksmart.com/og/YYYYad=753242267;YYYY qt=downloadantivirusfreeYYYYsubid=6990.191;YYYY rh=accurately-locate.com|http://buy.norton.com/YYYY|referrer	74.120.237.11	Search ad network correctly resolved
11	GET	302	http://buy.norton.com/YYYY**68998-6990.191**| www.accurately-locate.com/jump1?affiliate=6699& subid=191&terms=downloadantivirusfree& YYYY	166.98.228.51	Legitimate host name correctly resolved

Table 1: Example of organic search click hijacking: The victim intended to visit `free.avg.com` and is instead taken to a sponsored result from search ad network, `looksmart.com`, for `buy.norton.com`. The occurrence of pattern "YYYY" in the URL indicates a truncation for brevity sake.

work, `DoubleClick`, and publisher website victim visited, `eBay.com`, lose ad revenue that they were expecting to earn. This attack is stealthy from victim's perspective since the content on the website they visit is unchanged. Further details of the attack as described in Section 4.

2.2 Click Hijacking Attack

There are four different modes for this attack vector. We describe one here. The rest are outlined in Section 5. In this mode, the victim's click on an organic search engine's result is hijacked and the victim is redirected to another website. Table 1 shows the steps that ensue:

Step-1: The victim visits a search engine, `google.com`. It is resolved correctly through a malicious resolver. The victim enters search terms, "download free anti virus", which returns a set of search results.

Step-2: The victim clicks on an organic search result, `free.avg.com`, which is resolved correctly.

Step-3: The victim visits `free.avg.com` which is resolved correctly.

Step-4: The site, `free.avg.com`, attempts to load a Google Analytics script, `ga.js`. The malicious resolver mis-resolves the host name for this script, `www.google-analytics.com`, to inject a script called `123.php` into the parent document.

Step-5: The injected script, `123.php`, is loaded from a non-existent host name, `search2.google.com`, which is resolved by the malicious resolver to another malicious IP address. The script `123.php` sends the referrer, `www.google.com`, search terms entered by the victim, as well as the original site the victim clicked on to another non-existent host name, `search3.google.com`.

Step-6: The host, `search3.google.com`, has access to original search terms and website victim intended to visit, along with the referrer. The host makes a decision to hijack the click or not based on the value of the referrer. If the referrer indicates that the user came from a search engine, the click is hijacked. Otherwise, an empty response is returned instead. Since the referrer was `google.com`, the click is hijacked by returning an HTML frame that sends the victim to an attacker-controlled front-end website, `bulletindaily.com`.

Step-7: When the victim visits the front-end website, malicious resolver mis-resolves it to guard against external scrutiny just as in the case of ad replacement attack. The front-end website contains a form to submit search terms to another malicious IP, `65.60.9.238`.

Step-8: The IP, `65.60.9.238`, chooses to monetize the click by passing it through an affiliated fake search engine, `accurately-locate.com`, using a 302 redirect.

Step-9: The fake search engine, `accurately-locate.com`, has access to the victim's search terms. It uses them to generate a click on one of its sponsored results, `buy.norton.com`, which is retrieved from a search ad network, `looksmart.com`.

Steps-10-11: `lookSmart.com` redirects the victim to `buy.norton.com`. The referral field for both steps is `bulletindaily.com`.

In this example, the attackers monetized a user's click on an organic search engine's result by selling the click through an affiliate fake search engine. The page, `free.avg.com`, was loaded temporarily before being redirected to `buy.norton.com`. The attackers defrauded `avg.com` by stealing its traffic. Search ad network, `LookSmart`, and `Norton` were defrauded by simulating a click on `Norton's` textual ad and collecting CPC revenue from `LookSmart` through its affiliate fake search engine. We observed clicks being monetized through one of three kinds of intermediaries: ad networks, ad exchanges and affiliate marketing programs.

3 Identifying When Resolvers Lie

We started our investigation with two IP addresses of malicious resolvers in the 213.109.0.0/20 prefix which were given to us by a Trend Micro researcher involved in helping the FBI with Operation Ghost Click. We probed these malicious resolvers to infer when they lied about ad host names. Toward that goal, we visited top 3,000 websites according to Alexa [1] on May 11, 2011 and extracted ad URLs by running the captured HTTP traffic against the ad URL pattern matching list [6] used by Adblock Plus, a popular ad blocking tool. Since the starting point of most ads today are Javascripts or HTML URLs which subsequently include or generate URLs to other ad components, such as images, we focused only on Javascripts and HTML URLs to identify ad hosts. The resulting data set contained a total of 7,483 unique HTML and Javascript ad URLs which were delivered by 1,019 ad hosts. We queried the malicious resolvers

for these host names and applied the following two filtering heuristics to determine when a host name was mis-resolved. Note also that our filtering heuristics are intentionally generous, in that they look for any explanation that may justify a resolution returned by a malicious resolver as correct before labeling it as incorrect. Consequently, we might underestimate which ad host names malicious resolvers lie about but are unlikely to penalize a good resolution from a malicious resolver as incorrect.

- Heuristic 1: Resolution contains a valid IP address: For each ad host name resolved by a malicious resolver, this filter checks if any of the IP addresses were also returned by a good DNS resolver. Since DNS resolutions may be influenced by the geographical location of the resolver, such as when a content distribution network (CDN) is in use, or load balancing considerations, we gathered good DNS resolutions from 4,490 public resolvers around the world covering 74 countries [8]. Each public resolver in the list was queried for the `A` record of all ad host names multiple times until it stopped responding with new IP addresses. Specifically, if an IP address returned by a malicious resolver was returned by a public DNS resolver for any ad host name, this heuristic considers all IP addresses in that resolution to be good.

Running this heuristic on 2,952 IP addresses returned by the malicious DNS resolvers for the 1,019 ad hosts cut down the set of suspicious IP addresses by 90.5% in that 281 IP addresses remained unverified by this filter. These IP addresses corresponded to 96 host names.

- Heuristic 2: Suspicious IP returns a valid SSL certificate: This heuristic leverages that many ad networks support secure Web-based logins for their advertisers for tasks such as updates to their ad campaign and payment. To apply this heuristic, we established an HTTPS connection with each IP address of a valid resolution for the 96 host names remaining to be verified from Heuristic 1 in order to determine if the corresponding host names provided secure logins. We found that over 98% of the valid IPs corresponding to 62 host names returned a valid certificate. (The remaining 2% are attributable to connection failures or maintenance issues.) Upon examining the suspicious resolutions of the 62 host names, we found **8** suspicious IP addresses that did not return a certificate. We took this to indicate that these 8 IP addresses were malicious. These 8 malicious IPs corresponded to four host names. In fact, they were hosting an additional 23 host names, bringing the number of mis-resolved host names to 27. The list of mis-resolved host names included `ad.doubleclick.net`, `view.atdmt.com`, `tag.admeld.com` and `pagead2.googlesyndication.com`, each of which are popular ad hosts.

In analyzing the DNS records (`A`, `NS`, `SOA`) of these 8 malicious IPs, we observed common features. For example, we saw a typical TTL of 600 and `NS` & `SOA` records that were consistent.

4 Aspects of Ad Replacement

4.1 Ad Networks Defrauded

Section 3 helped identify 27 ad host names malicious resolvers were lying about. There were 1,277 Javascript and HTML ad URLs that belonged to them. In order to understand how the attackers were carrying out ad replacement, we setup a test machine to use a malicious resolver as its primary DNS resolver and visited each of the 1,277 ad URLs. The process helped identify how attackers were earning CPM revenue, as described in Section 2.1. We also learnt that the attackers had accounts with three ad networks shown in Table 2. We saw a total of four ad hosts from these networks and 34 different publisher IDs used by the attackers. The Table also shows the `CNAME` records for these host names, which we found by querying good DNS resolvers. The CNAMEs indicate that Redux Media and Xtend Media are both managed by Yahoo!.

Ad Network	Ad host names	CNAME	Publisher IDs
Redux Media	`ad.reduxmedia.com`	`ad.YM.com`	9
	`ads.reduxmediagroup.com`	`ib.adnxs.com`	23
Xtend Media	`ad.xtendmedia.com`	`ad.YM.com`	1
Clicksor	`ads.clicksor.com`		1

Table 2: Ad networks the attackers contracted with (YM stands for `yieldmanager`)

4.2 Operational Details

Out of the 1,277 ad URLs we tested, the attackers only successfully executed 782 URLs. In trying to answer why, we learnt a few interesting operational details about the attack:

- When unable to parse an ad URL, attackers loaded the original ad: We noticed that the size of the ad played an important role in determining whether an ad could be replaced or not. For example, in the ad URL shown in Figure 1(a), we can see that the size of the ad is `300x250` pixels which is clearly supported by `Xtend Media`, as evident from the completion of the attack chain. On the other hand, if the attackers encountered an ad size they could not support, they returned an empty response. As a concrete example, using the same ad URL as in Figure 1(a) with a modification of the size variable to `300x50` would cause the attackers to return an empty response. We found they supported 18 common size variations such as `160x600` and `728x90` while sizes `120x60` and `150x150` were not supported.

In cases where the attackers either did not support the ad type or could not extract size from the ad URL, the initial mis-resolution of the ad host redirected the victim to the intended ad server so that the original ad could be displayed. For example, if the path in the ad URL shown in Figure 1(a) was modified slightly by replacing `adi` with `ad`, as `http://ad.doubleclick.net/ad/ebay.ebayus.homepage.cchp` , the attackers would redirect the victim to `http://ad2.doubleclick.net/ad/ebay.ebayus.homepage.cchp`. Note that this modification is specific to ad host, `ad.doubleclick.net`. Our version of the edited URL specifies that the ad request is for an image (denoted by `/ad/`) not an iFrame (denoted by `/adi/`), as was the case with the original URL in Figure 1(a) which was supported by the attackers. Upon redirection to `ad2.doubleclick.net`, the attacker correctly resolves the host name to allow the originally intended ad to be loaded. The redirection helps the attackers escape detection from the ad network whose ad they were replacing. We observed the same redirection behavior for other ad hosts as well.

```
http://pagead2.googlesyndication.com/pagead/show_ads.
js?1306433500880
```

Figure 2: An example ad URL without size information. The identifier at the end is not size related.

- Attackers missed ad replacement opportunities: While examining URLs, we noticed cases where the attack-

ers redirected the victim to the original correctly resolved ad host, causing them to miss out on additional profit. Specifically, we found that ad sizes were parsed in a case-sensitive manner causing them to miss for example `300X250`, where the attackers were parsing only sizes with a small 'x'. Additionally, they were failing to recognize minor variations in URL paths and minor ad size variations where they were only replacing ads with exact sizes. For example, an ad URL with a size of `300x251` was not replaced with an ad of size `300x250`, where the later was supported by the attackers.

5 Modes of Click Hijacking

There are four different modes of the click hijacking attack, shown in Figure 3. We described one in Section 2.2. Here, we outline them all.

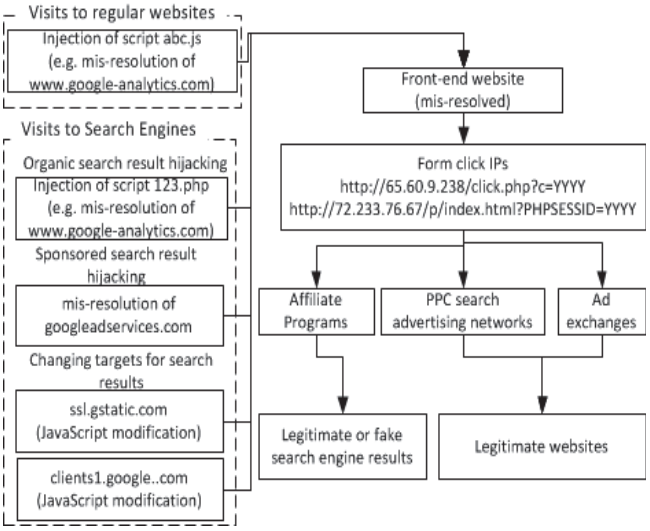


Figure 3: The four modes of the click hijacking attack

5.1 Visits to Search Engines

Most search engines display two types of results: *organic* and *sponsored*. While organic results appear because of their relevance to the search terms entered by a user, sponsored results are ads from advertisers who have contracted with the search engines. Search engines often attempt to display the most relevant sponsored results in the interest of luring visitors into clicking on them since clicks generate money for them.

In three of the modes of click hijacking, the attackers exploited the fact that most Internet users visit web pages through search engines and monetized their clicks. Broadly speaking, the attackers were either changing the targets of results displayed by a search engine or were leaving the targets as is but were hijacking clicks victims made on pages they visited through search engines. We outline each of the three modes below.

5.1.1 Organic search result hijacking

In the first mode, the attackers were monetizing clicks on organic search results, as detailed in Section 2.2. They were monetizing clicks through the injection of a server-side script, `123.php`, which was injected any time the victim clicked on an organic search result page containing a Javascript. We saw concrete instances of the injection on pages with the Google Analytics Javascript, `ga.js`, and when the attacker successfully replaced an ad involving a Javascript.

After hijacking a click, the victim was always directed to a front-end website which contains a form to submit search terms to one of a set malicious IP addresses. The form was submitted using the same search terms as the ones entered by the victim in the search engine. (Further testing revealed cases where the attacker was changing the keywords. In one instance, the keywords "download antivirus free" were changed to "best digital cameras".) We refer to this set of IP addresses as *form click* IPs in this paper. The form click IP addresses decided on the method of the click monetization by redirecting the user to one of the following options:

- Monetize through a pay-per-click (PPC) search advertising network: Many search engines and web portals with search functionality display text-based sponsored ads from PPC search advertising networks. A click on one of those ads redirects the user to the PPC search ad network, who does accounting based on the publisher's ID and search terms. We witnessed clicks being sold to PPC search ad networks such as `looksmart.com`, `admanage.com`, `adknowledge.com` and `dsidemarketing.com`.

- Monetize through an ad exchange: Ad exchanges are technology platforms that facilitate bidded buying and selling of online ads from multiple ad networks. We saw instances where a click on `free.avg.com` was sold through the `bidsystem.com` ad exchange to advertiser, `stopzilla.com`. We also witnessed other ad exchanges exploited in a similar way such as `qualibid.com`.

- Monetize via affiliation with search engines: In this form of monetization, the attackers set affiliate accounts with search engines and led victims to it. The latter displayed results for the keywords entered by the victim. The search engines paid the attackers based on their affiliate IDs. We found two types of search engines being used, legitimate and fake. The *fake search engines* were not directly controlled by the attackers. They displayed sponsored results through PPC search ad networks and clicks on these sponsored results were used either as-is or redirected to a malicious site depending on the affiliate who sent them the traffic. We also found instances using legitimate search engines where our clicks on `free.avg.com`, in the example described in Section 2.2, led to a set of search results displayed on a legitimate search engine, `star.feedsmixer.org`. We witnessed the attacker exploiting other search engines such as `infomash.org` and `info.com`, the same way.

5.1.2 Sponsored search result hijacking

In the second mode of the click hijacking attack, the attackers were hijacking clicks on sponsored search results in a manner similar to that for organic search results. Specifically in the case of Google, clicks on the sponsored search results were processed through `googleadservices.com`. The attackers were mis-resolving this host name to a new, dedicated malicious IP address which then ultimately led to a dedicated front-end website, `relited.com`. This website had a couple of press releases that appeared to be propagandas to establish its legitimacy in case ad networks investigated it for the clicks attackers were cashing. Once the victim reached the front-end website, it was redirected to one of the form click IPs to monetize this click in one of the three ways described in Section 5.1.1.

5.1.3 Changing targets for search results

In contrast to hijacking actual clicks on organic or sponsored search results, in the third mode of the attack, the attackers

employed an alternative mode of hijacking human clicks. In this mode, they changed the targets for sponsored as well as organic search results. We explored this attack in detail only in the context of the Google search engine but have preliminary evidence that the attackers were exploiting other popular search engines similarly. The attack essentially worked by mis-resolving the host names for two of the Javascripts used by Google in displaying search results. The first script is loaded from `clients1.google.com` and is used for autocompletion during typing of keywords. The second is loaded from `ssl.gstatic.com`, which is a Google host dedicated to static content, including images, Javascripts and CSS files. The attackers mis-resolved each of these two host names to two dedicated malicious IP addresses each and changed the functionality of these scripts by adding a snippet of code which resets click event handlers, such as `onclick` and `onmousedown` to cancel out the action originally set by Google and changes where the links in the returned HTML document lead. The modified links lead to non-existent host names with an argument list containing the original link returned by Google, keywords and referrer fields. Once the victim reached there, it was redirected to one of the form click IPs to monetize this click in one of the three ways described in Section 5.1.1.

5.2 Visits to Regular Websites

In the fourth and final mode of the click hijacking attack, the attackers were monetizing visits to regular websites that may not have originated at a search engine. This mode of attack was executed any time they managed to inject a Javascript, `abc.js`, using the same method of injection for the `123.php` script. Correspondingly, upon injection of the script, the attackers choose between loading `123.php` and `abc.js` randomly.

The script, `abc.js`, set two types of event handlers: 1) body onclick and 2) `<a>` link onclick. The script exploited only the first click of any of these events to avoid frustrating the victim to the extent that they would take immediate steps to get their machines cleaned up. As it is the mean time to clean the DNS-changer malware was 6-12 days [20]. The goal of this script was to open a new window in response to the click event. The new window belonged to non-existent host names from reputable domains and the attackers generated these host names based on what type of Javascript led to the injection of the malicious script. For example, if `abc.js` was injected through the Google Analytics Javascript, `ga.js`, then the new window was loaded from `search.google-analytics.com`. Since these host names were resolved by the attacker-controlled malicious resolvers, the attackers could choose any. Carrying the `search.google-analytics.com` example further, a visit to the URL in the new window ultimately led to one the dedicated front-end websites controlled by the attackers. The front-end websites were mis-resolved as before. The front-end websites dedicated to the `abc.js` script resolved to a group of three malicious IP addresses. Ultimately, the click was monetized in one of the three ways described in Section 5.1.1.

6 Attack Infrastructure

The attack infrastructure had three components. The first were the malicious resolvers. The second were the malicious websites the attackers used to earn CPM and CPC revenue. The third were the malicious IP addresses used in executing the attack chains. While our experiments revealed some of each, we suspected that there were more. In this section, we attempt to find the rest.

6.1 Malicious Resolvers

We started this study with the IP addresses of two malicious resolvers. Upon searching for online articles reporting on the behavior of malicious DNS resolvers [18, 32, 26], we found several IP addresses belonging to six IP prefixes reported to be acting malicious or used by a DNS changer malware. To find additional malicious resolvers in these prefixes, we scanned each IP in these prefixes and queried for an `A` record for `ad.doubleclick.net`, a popular ad host targeted by malicious resolvers. A resolver is considered malicious if it mis-resolves `ad.doubleclick.net`. Incorrect resolution is determined using the filtering heuristics discussed in Section 3.

Table 4 shows the total number of malicious resolvers we found and the numbers we found reported in online articles. (Details about the owners of these prefixes were derived using the Hurricane Electric BGP Toolkit [5]). A high number of malicious resolvers were found in the IP prefixes in Russia and Ukraine. In contrast, only one resolver was found across the two prefixes we scanned belonging to the U.S. This is likely a result of malicious resolver migration as was reported from prefix 85.255.112.0/24 to 99.198.101.1/19 in 2008 [26].

IP Prefix	Country	Organization	AS	Malicious Resolvers	
				Reported	Found
213.109.0.0/20	Russia	Lipetskie Kabelnie	AS42368	2	166
93.188.160.0/24	Ukraine	Promnet Ltd.	AS36445	1	211
93.188.161.0/24				1	240
93.188.162.0/23				1	422
69.31.52.0/23	US	Pilosoft	AS26627	48	1
99.198.101.0/19		Singlehop	AS32475	51	0

Table 4: Malicious resolvers found in each IP prefix scanned

6.1.1 Behavior seen at .com/.net

Next, we examined the behavior of malicious resolvers in the query traffic seen at Verisign's `.com` and `.net` DNS Top Level Domain (TLD) infrastructure, and its instances of the global DNS root zone. Verisign services over a billion DNS queries per day, hosts well over one hundred million domain names, and because of `.com`'s and `.net`'s popularity, one could argue that almost every heavily used DNS resolver will eventually send queries to this infrastructure [33]. Additionally, Verisign's infrastructure is composed of multiple sites around the world and on several continents. Thus, DNS resolvers that follow the *pinning and polling* behavior described in [33] would likely issue portions of their query load to multiple servers, but ultimately pin themselves to the topologically closest site for the bulk of their queries.

We examined data taken on October 20th, 2011, prior to the take-down of attack infrastructure. It came from roughly half of all of this provider's global sites, which included all of those within the US and several in Europe. Any resolver looking to resolve a host name belonging to `.com` or `.net` would contact these servers in situations where it does not have the host name cached locally. While our view was limited to roughly half of the global sites, it offered us interesting insights into the attack infrastructure. The first observation we made was that none of the known malicious resolvers sent any queries to the TLD servers. This

IP	ASN	BGP Prefix	Owner	Websites Hosted	Use
69.31.52.10	AS26627	69.31.52.0/23	Pilosoft	31	Front-end websites
69.31.42.125	AS26627	69.31.40.0/21	Pilosoft	189	
69.31.42.126	AS26627	69.31.40.0/21	Pilosoft	46	
67.29.139.153	AS3356	67.29.136.0/21	Level 3 Communications	147	Fake search engines
63.209.69.109	AS3356	63.209.64.0/21	Level 3 Communications	13	

Table 3: Valid resolutions of malicious websites. The number of front-end websites adds up to 266 but only 263 of these were unique.

seemed counter-intuitive since we knew that these resolvers returned correct resolutions for most host names queried. When we looked for any queries from the known malicious prefixes, we found 13 IP addresses that were different from the known malicious resolvers. Clearly, the malicious resolvers we found were simply *DNS forwarders*. In fact, many infrastructure providers configure their DNS resolvers to forward their client queries to a more centralized DNS resolver to benefit from caching, better provisioning, etc. Interestingly, 10 of the forwarders were from the Russian prefix and three from the one U.S. prefix. None queried for `ad.doubleclick.net`, the most commonly misdirected ad host. Further, the forwarders queried for several thousand host names during the course of a day but the peak query rate observed from any forwarder was 500 queries per minute, which is lower than one would expect from busy resolvers [33]. Finally, the Russian forwarders were busier and exhibited little in the way of diurnal patterns, possibly indicating the diversity of victims around the globe. However, the forwarders in the US exhibited a distinct peak at 10am.

6.2 Malicious Websites

The ad replacement and click hijacking attacks use front-end websites as a disguise to convince the defrauded ad networks that they serve useful content to Internet users. Additionally, in click hijacking, we observed the use of fake search engines. We found a total of 42 front-end websites and 43 fake search engines during our experiments. In order to expose more malicious websites and explore their features, we devised an iterative algorithm whose basic idea was to take known IP addresses from good resolutions of known malicious websites and find what host names (i.e. reverse look up) they corresponded to using the Hurricane Electric BGP Toolkit [5], which provides a list of host names whose resolutions have returned the IP under consideration. If previously unknown host names were found, we resolved them through malicious resolvers. If they were found to be mis-resolved using the filters we used in Section 3, they were strong candidates for front-end websites. We repeated the same iterative process using the incorrect resolution of the front-end websites and correct resolutions of the fake search engines (since they were never mis-resolved) on the data set of HTTP transactions used in Section 7. This iterative process resulted in a larger set of malicious websites as illustrated in Table 3. Now, we describe their characteristics.

6.2.1 Front-end websites

The iterative process described earlier to identify more front-end websites expanded the list to 263 websites. We manually examined the 42 we found in our experiments. The content on the front-end websites consisted of scrapped content grouped into various categories. In terms of variety of content of these front-end websites, we saw a wide variety. For example, world news (four variations), software news,books for sale, music sites (two variations), video sites (two variations), adult products for sale and others. When these sites offered products for sale, they would eventually link to a legitimate vendor of those products. The front-end websites were all designed to look like web portals with only a few variations in the style sheets for the layouts but with vastly different color schemes. Many of them had a search box that typically only returned sponsored results. They all used one type of "Contact Us" form. Also, the privacy policies listed were fundamentally similar. Despite the dynamic nature of content, the sites hardly made use of client-side scripting, implying that the attacker was deciding on content at its own servers.

The front-end websites used in click hijacking differed from those used in ad replacement in that the latter served ads on their main pages to convince the ad networks they were defrauding that they were indeed serving their ads on those sites. Furthermore, each front-end website had a different sub domain/URL structure specific to the ad networks they were defrauding for management purposes.

The valid resolutions of front-end sites corresponded to three IP addresses belonging to three organizations, as shown in Table 3. Looking at their registration information through *whois* revealed interesting details. The registrant for all but five was "RegName.biz". The rest were registered under "Regtime Ltd". The registration department was either "Advertising network" or the name of the front-end domain followed by "Information department". There were two variants of registrants addresses; one in New York, US while the other was in Benhavn, DK. This homogeneity could have been exploited to find them all.

6.2.2 Fake Search Engines

Even though the fake search engines were not explicitly owned by the attackers, they were used to facilitate the click hijacking attack. Hence, we considered them to be a part of the attack infrastructure. Our iterative process revealed a total of 160 fake search engines.

We manually examined the 43 fake search engines we found during our experimentation and found that 33 of them had only one search text box and search button with links to privacy policy, advertisers, publishers and About Us. They were essentially using one template by Free CSS Templates [3] but with different colors. The rest included news feeds and a weather plugins in addition to the main search text box and button. They were all registered with the same registrar, "Dotster" [2] with two variations of registrant address. The first was in Vancuver, WA while the other was in Encino, CA. Finally, their contact email addresses for inquiries were all of the form "bizdev@fake search engine.com" and "advertise@fake search engine.com". Table 3 shows the two valid IP addresses all the fake search engines resolved to and that both these IP addresses belong to one organization.

6.3 Malicious IP addresses

During the course of our investigations of ad replacement and click hijacking, we found a total of 17 malicious IP addresses where all but two them were used to mis-resolve var-

Attack Type	IP	ASN	BGP Prefix	Owner	Mis-Resolved Valid Hosts/Comments
Ad Replacement	216.180.243.10	AS3595	216.180.224.0/19	Global Net Access, LLC	Ad hosts (ex:"ad.doubleclick.net")
	65.254.36.122		65.254.32.0/20		
	75.102.23.112	AS23352	75.102.0.0/18	Server Central Network	Ad host "pagead2.googlesyndication.com"
	205.234.231.37		205.234.128.0/17		
	205.234.201.230				
Click hijack (visits to search engines)	67.210.15.16	AS36445	67.210.14.0/23	Internet Path	Sponsored results from "googleadservices.com"
	67.210.14.[53-54]				Organic results via script `123.php`
	67.210.15.37 67.210.15.39				Modified search results from "clients1.google.com"
	67.210.15.[70-71]				Modified search results from "ssl.gstatic.com"
Click hijack (visits to regular websites)	75.102.23.111	AS23352	75.102.0.0/18	Server Central Network	Ad replacement or injection of `abc.js` via Google Analytics' script, `ga.js`
	205.234.231.39		205.234.128.0/17		
	205.234.201.229				
Click hijack (form click IPs)	65.60.9.[235-238]	AS32475	65.60.0.0/18	SingleHop, Inc.	Form click IPs
	72.233.76.[66-70]	AS22576	72.233.0.0/17	Layered Technologies	
	94.102.60.6	AS29073	94.102.48.0/20	Ecatel LTD	
Blacklisting defense	67.210.15.38	AS36445	67.210.14.0/23	Internet Path	"safebrowsing.clients.google.com"
Ad replacement & Click hijacking	67.210.14.254				Parked domains
Phishing	67.210.14.189				Sub domains of adult sites (ex:"join.cheerleaderauditions.com")
	216.163.137.61	AS14068	216.163.136.0/23	Playboy	
	62.141.56.61	AS31103	62.141.48.0/20	Keyweb AG	

Table 5: Summary of all malicious IP addresses found

ious ad hosts and search engine host names. The remaining two were *form click* IPs used to simulate form clicks on attackers' front-end sites.

Using the data set of HTTP transactions used in Section 7 we searched for host names corresponding to the 17 known malicious IP addresses. If new host names were found, we checked for signatures of ad replacement and click hijacking attacks. Also, we checked if the resolution against malicious resolvers conformed to the characteristics observed in Appendix B.

This process resulted in 30 total malicious IP addresses. Their functionality, along with the ISPs they belonged to, is summarized in Table 5. Interestingly, a few of the malicious IPs carried out functionality not observed in our experiments. One malicious IP was used to mis-resolve a host name for Google's Safe Browsing API, which blacklists phishing and malware domains [4]. The attackers were returning an empty response to bypass blacklisting. Another three were used to run a phishing attack on adult sites as described by Trend Micro [23]. We also found an IP address that was used to mis-resolve parked domains so attackers could divert their traffic to their own ad pages.

Although the attack infrastructure was taken down [27, 34, 36], we found the IP addresses shown in red color in Table 5 to be still up and functioning for a couple of weeks after the take down. Incidentally, all of them belong to one ISP, "Server Central Network".

7 Impact of the Ad Fraud Scheme

To understand the impact of ad replacement on real users, we placed a network monitor on a Broadband Remote Access Server (BRAS). The BRAS we used is an aggregation point for Digital Subscriber Lines (DSLs) for a large Tier 1 ISP's customers located in the United States (U.S.) and serves approximately 17,000 active broadband subscribers. Our analysis was conducted using aggregated data collected from all HTTP traffic transiting this particular BRAS on 2/15/2011. We only had access to HTTP headers and no data packets. The privacy of the subscribers was preserved since the dynamic IP addresses were not mapped to individual households and the study focused on the aggregate traffic across all the subscribers. Further, since query strings in URLs can contain sensitive information, we did not examine the query strings in order to protect subscriber privacy.

Our data contained the source IP, the destination IP, the URL, the response status code, the user agent, and the referrer field for each HTTP request. We used it to reconstruct user sessions, to aid in estimating the occurrence of each type of attack. To find user sessions, we divided the traffic by source IP, which could include multiple users, and then divided each of the resulting sets by the transmitted user-agent string to separate out sessions from the same browser. Each of the user-agent/source IP sets of traffic was then grouped into sessions by identifying time gaps in the traffic. Subsequently, we extracted sessions containing transactions to the known malicious IPs and chained them via the HTTP referrer fields. Chains with malicious IPs were then checked to determine entry points for the attacks. We additionally checked all chains for known entry points in order to find other chains that used unknown malicious IPs.

A summary of the results is in Table 6. *We found 37 infected user lines in our data.* From these lines, we observed 2,811 possibilities for the ad replacement attack, of which 1,800 are successful. Also, we found 144 click hijacking attacks. Of the latter, 111 were organic search result hijacks done by injecting the `abc.js` script through Google Analytics. 6 were hijacks of sponsored search results and the remaining 27 were cases where the attackers changed target URLs for search results via mis-resolution of `clients1.google.com`. We note that there were many more calls to `abc.js` than the successful hijacks reported in Table 6: 2,334. However, these attack chains were not completed. There are several possible explanations to this. First, the attacker may have a strategy to limit the number of click hijacks per machine in a given time-frame, to reduce the footprint of the fraud. Second, it is also possible that there was an undiscovered attack mode. Third, the attacker's infrastructure may have changed between data collection and the mapping of the attack chains. Finally, we note that two of the click hijacking attack modes shown in Figure 3, namely, those that start with the injection of `123.php` on a non search engine website and mis-resolution of `ssl.gstatic.com` were not observed in our data likely because they modes were newly added.

This data also revealed that *257 legitimate content publishers lost revenue due to this fraud scheme.* 17 lost ad revenue for at least 20 ad impressions and clicks each. Also, *we saw 21 different ad hosts whose ad networks lost revenue, with* `ad.doubleclick.net` *being the most popular. They belonged to 20 ad networks.*

Total subscriber lines	17,000
Total HTTP requests	55,024,300
Infected lines	37
HTTP requests from infected lines	295,699
Possible ad Replacement attacks	2,811
Actual ad replacement attacks	1,800
Click hijacking attacks	144
Organic search result hijacks	111
Sponsored search result hijacks	6
Hijacks via changed search result targets	27

Table 6: Impact of ad fraud on DSL subscribers of a Tier 1 ISP

While we realize that our data is not a representative of the world or even the United States (U.S.), we extrapolate the numbers to compare our estimations with that of Operation Ghost Click [27, 34, 36]. There are almost 86 million subscription lines in the U.S. [9]. An extrapolation from our data would suggest that 186,574 subscription lines were infected. Extrapolating it world-wide, there would be 1,176,795 infected lines, as there are 540 million subscription lines world wide. Note that there may be multiple computers per subscription line. Member countries of the Organisation for Economic Co-operation and Development (OECD) typically have three times as many Internet users as they do subscription lines, determined by comparing the number of subscription lines according to [9] and the number of Internet users according to [10]. Accounting for this, we estimate the affected subscription lines to be 3.53 million, which is somewhat consistent with the estimate of 4 million given by the FBI.

8 Potential Mitigation Strategies

Ads are an important component of the Internet economy. Consequently, while attackers find them to be an attractive revenue stream, ad networks take steps to combat the threat [25, 28]. The ad networks employ various techniques to shield themselves and their advertisers from ad fraud. For example, many audit publisher websites to ensure that they indeed serve the content they claim to be serving. The attackers of the ad fraud scheme discussed in this paper provisioned their front-end websites to render this defense ineffective since the correct resolutions of their front-end websites displayed web pages that looked legitimate. Along the same lines, ad networks also try to match traffic to publisher websites with respect to their page rankings. Further, historical information about publishers is also used to detect sudden changes in ad traffic patterns. We lack data to judge if the front-end websites and fake search engines used by the attackers would be caught under these tests. Next, we discuss possible methods for detecting ghost clicks. Each of these methods rely on the special characteristics of the ad fraud scheme we studied.

- Serving bluff ads: Researchers in [24] explored the use of bluff (bait) ads which are fake ads instrumented to detect illegitimate clicks. They can also be used to detect misdirected human clicks of the types attackers in our ad fraud scheme exploited.

- Finding fake publisher websites: We noted homogeneity among front-end websites and fake search engines. As ad networks discover fraudulent publisher websites, they can leverage homogeneity to find other sister websites.

- Using HTTP with integrity: A simple solution to prevent misdirection is to use a partially secure version of HTTP, namely HTTPI (HTTP with integrity) [13], offers a practical solution advertisers can now employ.

- Monitoring and scrutinizing unexpected DNS resolvers: This defense attacks the key component of the ad fraud we studied – that victims' DNS resolvers were incorrectly configured to attacker-controlled machines. ISPs can monitor DNS requests and responses traversing their organizational borders to find unexpected DNS resolvers. Once found, their responses can be scrutinized using filtering heuristics similar to ones we described in Section 3.

- Identifying accounting discrepancies: Many websites use equivalents of Google Analytics to keep track of the visitors to their sites. Publishers and ad networks can employ better auditing to detect discrepancies.

9 Related Work

Ad fraud has recently been studied. Authors in [31] and [16] reverse engineered clickbots, `Fiesta`, `7cy` and `Clickbot.A` to understand their behavior. Their methodology to understand bot modus operandi is comparable to ours. In a complementary work, Miller et al. [35] investigated a clickbot and used that understanding to study how ad networks can identify and combat click fraud. In the same vein, Dave et al. [17] proposed a methodology advertisers could use to measure click spam rates on their ads. These works differ from ours in that they focus on clickbots.

Zhang et al. conducted a study in which they bought traffic and explicitly located the presence of human clickers as well as clickbots [42]. Their work proves the existence of human clickers in ad fraud while clickbots were well known earlier. Our work expands the landscape of ad fraud to include misdirected human clicks. Since the only malware involved in the fraud we investigated is a DNS changer malware, our work demonstrates the power of DNS misdirections in creating novel ad fraud schemes. Trend Micro researchers helped the FBI in Operation Ghost Click and were investigating the infrastructure we dissect here before us and helped bootstrap our work. Even though their blog provides an account of the takedown and botnet operation [22], to our knowledge, we are the first to document the anatomy of both attacks rigorously.

Inflight modification of Web content has been studied by Zhang et al. in [41]. While the topic of inflight modification may appear orthogonal to ad fraud, the authors discovered that much of the observed modification was attributable to malicious DNS resolvers and ad fraud. These resolvers were sometimes compromised resolvers at the ISPs and sometimes resolvers victims of DNS changer malware were pointed to. This work does not look into how the observed ad fraud was carried out but emphasizes the need to watch DNS resolvers closely.

The malicious resolvers that were central to the ad fraud scheme studied in this paper were open resolvers. In [14], Dagon et al. examined the entire IPv4 space to find open DNS resolvers in the Internet and then to determine when they lied. This is a similar problem to one we encountered while developing heuristics to conclude when malicious resolvers lied. Dagon et al. determined correctness of a resolver response by comparing net blocks of the answers, and then by visiting incorrect resolutions. We found this approach to be problematic in our context. This is because ad hosts typically use CDNs and the answers from geographically displaced CDN servers will legitimately be from different net blocks. Further, ad hosts return non-deterministic ad URLs,

ruling out re-visitation. Zdrnja et al. captured DNS replies at the University of Auckland Internet gateway in order to detect unusual DNS behavior [39]. They focused on typo squatting, domains abused by spammers and fast-flux domains. Our work is different in that we investigate all types of incorrect resolutions whether they were for search engines, ad hosts, parked domains or hosts for antivirus.

The network of victims infected with DNS changer malware can loosely be referred to as a botnet. There have been numerous published reports of botnet takedowns and infiltrations [29, 12]. They typically focus on botnets involved in spam, distributed denial-of-service (DDoS) attacks or phishing. The difference in purpose in our case rules out the need for a traditional botnet command-and-control (C&C).

10 Conclusions

Our work highlights the increasing sophistication of ad fraud schemes. Specifically, the attackers were making creative use of misdirected clicks from real human beings in order to steal CPM and CPC revenue from 20 different ad networks, all the while fooling over a dozen other ad networks that they earned revenue for these clicks and impressions at 423 publisher websites. A critical enabler for the documented attacks was DNS changer malware, which misled victims to resolvers located in Eastern Europe instead of resolvers of their choice, such as those belonging to their ISPs. Our work points to the need to closely monitor DNS resolvers Internet users query.

11 References

[1] Alexa: The web information company. http://www.alexa.com/.

[2] Dotster Inc.: Domain registration and website hosting. http://www.dotster.com/.

[3] Free CSS templates. http://www.freecsstemplates.org/.

[4] Google safe browsing API. http://code.google.com/apis/safebrowsing/.

[5] Hurricane Electric BGP toolkit. http://bgp.he.net/.

[6] The official easylist website. https://easylist.adblockplus.org.

[7] Trend Micro threat encyclopedia. http://threatinfo.trendmicro.com/vinfo/virusencyclo/default5.asp?VName=TROJ5FDNSCHANG12EBM&VSect=P.

[8] Ungefiltert surfen mit freien nameservern. http://www.ungefiltert-surfen.de/.

[9] Total fixed and wireless broadband subscriptions by country. Organization for Economic Co-operation and Development, 2010.

[10] Central Intelligence Agency World Factbook, 2011.

[11] How does Google detect invalid clicks? http://adwords.google.com/support/aw/bin/answer.py?hl=en&answer=6114, 2011.

[12] Cho, C. Y., Caballero, J., Grier, C., Paxson, V., and Song, D. Insights from the inside: A view of botnet management from infiltration. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)* (2010).

[13] Choi, T., and Gouda, M. HTTPI: An HTTP with integrity. In *International Conference on Computer Communications and Networks (ICCCN)* (2011).

[14] Dagon, D., Provos, N., Lee, C. P., and Lee, W. Corrupted DNS resolution paths: The rise of a malicious resolution authority. In *ISOC Network and Distributed Security Symposium (NDSS)* (2008).

[15] Daswani, N., Mysen, C., Rao, V., Weis, S., Gharachorloo, K., and Ghosemajumde, S. *Crimeware: Understanding New Attacks and Defenses*. Addison-Wesley Professional, 2008, ch. Online Advertising Fraud.

[16] Daswani, N., and Stoppelman, M. The anatomy of Clickbot.A. In *USENIX Workshop on Hot Topics in Understanding Botnets* (2007).

[17] Dave, V., Guha, S., and Zhang, Y. Measuring and fingerprinting click-spam in ad networks. In *ACM SIGCOMM* (2012).

[18] Eckman, B. Significant rogue DNS activity to 85.255.112.0/22. http://lists.sans.org/pipermail/unisog/2006-November/026937.html.

[19] F-secure. Trojan:osx/dnschanger. http://www.f-secure.com/v-descs/trojan_osx_DNSchanger.shtml.

[20] Feike Hacquebord. Making a million, part two: The scale of the threat. http://blog.trendmicro.com/?p=27054, 2010.

[21] Florio, E. DNS pharming attacks using rogue DHCP. http://www.symantec.com/connect/blogs/DNS-pharming-attacks-using-rogue-dhcp, December 2008.

[22] Hacquebord, F. Esthost taken down - biggest cybercriminal takedown in history. http://blog.trendmicro.com/esthost-taken-down-biggest-cybercriminal-takedown-in-history, November 2011.

[23] Hacquebord, F., and Lu, C. Rogue domain name system servers part 2. http://blog.trendmicro.com/rogue-domain-name-system-servers-part-2, September 2007.

[24] Haddadi, H. Fighting online click-fraud using bluff ads. In *SIGCOMM Computer Communications Review (CCR)* (2010).

[25] Harvey, S. Invalid clicks and online advertising. http://www.google.com/doubleclick/pdfs/DoubleClick-04-2007-Invalid-Clicks-and-Online-Advertising.pdf, April 2007.

[26] Jose, N. Rogue DNS servers on the move. http://asert.arbornetworks.com/2008/11/rogue-dns-servers-on-the-move/.

[27] Krebs, B. Biggest cybercriminal takedown in history. http://krebsonsecurity.com/2011/11/malware-click-fraud-kingpins-arrested-in-estonia/.

[28] Kshetri, N. The economics of click fraud. In *IEEE Symposium on Security and Privacy* (2010).

[29] Leder, F., Werner, T., and Martini, P. Proactive botnet countermeasures – an offensive approach. In *Cooperative Cyber Defence Centre of Excellence* (2009).

[30] McAfee. New DNSChanger trojan hacks into routers. http://www.trustedsource.org/blog/42/New-DNSChanger-Trojan-hacks-into-routers, June 2008.

[31] Miller, B., Pearce, P., Grier, C., Kreibich, C., and Paxson, V. What's clicking what? Techniques and innovations of today's Clickbots. In *SIG SIDAR Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)* (2011).

[32] Mosuela, L. Search engine redirection malware, how it works (and how to fix it). http://blog.commtouch.com/cafe/malware/how-it-works-search-engine-redirection-malware/.

[33] Osterweil, E., McPherson, D., DiBenedetto, S., Papadopoulos, C., and Massey, D. Behavior of DNS' top talkers, a .com/.net view. In *Passive and Active Measurement Conference (PAM)* (2012).

[34] Sengupta, S. 7 charged in web scam using ads. http://www.nytimes.com/2011/11/10/technology/us-indicts-7-in-online-ad-fraud-scheme.html?_r=2.

[35] Stone-Gross, B., Stevens, R., Zarras, A., Kruegel, C., Kemmerer, R., and Vigna, G. Understanding fraudulent activities in online ad exchanges. In *ACM/USENIX Internet Measurement Conference (IMC)* (2011).

[36] United states district court southern district of New york. Sealed indictment. http://www.wired.com/images_blogs/threatlevel/2011/11/Tsastsin-et-al.-Indictment.pdf, November 2011.

[37] Zdrnja, B. DNS Trojan for the Mac in the wild. http://isc.sans.edu/diary.html?storyid=3595, November 2007.

[38] Zdrnja, B. Rogue DHCP servers. http://isc.sans.edu/diary.html?storyid=5434, December 2008.

[39] Zdrnja, B., Brownlee, N., and Wessels, D. Passive monitoring of DNS anomalies. In *SIG SIDAR Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)* (2007).

[40] Zeltser, L. An overview of the freevideo player Trojan. http://isc.sans.org/diary.html?storyid=1872, 2006.

[41] Zhang, C., Huang, C., Ross, K. W., Maltz, D. A., and Li, J. Inflight modifications of content: Who are the culprits? In *USENIX Large-Scale Exploits and Emerging Threats (LEET)* (2011).

[42] Zhang, Q., Ristenpart, T., Savage, S., and Voelker, G. M. Got traffic? An evaluation of click traffic providers. In *WICOM/AIRWeb Workshop on Web Quality (WebQuality)* (2011).

Permission Evolution in the Android Ecosystem

Xuetao Wei, Lorenzo Gomez, Iulian Neamtiu, Michalis Faloutsos
Department of Computer Science and Engineering
University of California, Riverside
{xwei, gomezl, neamtiu, michalis}@cs.ucr.edu

ABSTRACT

Android uses a system of permissions to control how apps access sensitive devices and data stores. Unfortunately, we have little understanding of the evolution of Android permissions since their inception (2008). *Is the permission model allowing the Android platform and apps to become more secure?* In this paper, we present arguably the first long-term study that is centered around both permission evolution and usage, of the entire Android ecosystem (platform, third-party apps, and pre-installed apps). First, we study the Android platform to see how the set of permissions has evolved; we find that this set tends to grow, and the growth is not aimed towards providing finer-grained permissions but rather towards offering access to new hardware features; a particular concern is that the set of `Dangerous` permissions is increasing. Second, we study Android third-party and pre-installed apps to examine whether they follow the *principle of least privilege*. We find that this is not the case, as an increasing percentage of the popular apps we study are overprivileged. In addition, the apps tend to use more permissions over time. Third, we highlight some concerns with pre-installed apps, e.g., apps that vendors distribute with the phone; these apps have access to, and use, a larger set of higher-privileged permissions which pose security and privacy risks. At the risk of oversimplification, we state that the Android ecosystem is not becoming more secure from the user's point of view. Our study derives four recommendations for improving the Android security and suggests the need to revisit the practices and policies of the ecosystem.

1. INTRODUCTION

The popularity of the Android platform is driven by feature-rich devices, as well as the myriad Android apps offered by a large community of developers. Furthermore, smartphones have become an integral part of daily lives, with users increasingly relying on smartphones to collect, store, and handle personal data. This data can be highly privacy-sensitive, hence there are increased concerns about the security of the Android ecosystem and safety of private user data [11].

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

To ensure security and privacy, Android uses a permission-based security model to mediate access to sensitive data, e.g., location, phone call logs, contacts, emails, or photos, and potentially dangerous device functionalities, e.g., Internet, GPS, and camera. The platform requires each app to explicitly request permissions up-front for accessing personal information and phone features. App developers must define the permissions their app will use in the `AndroidManifest.xml` file bundled with the app, and then, users have the chance to see and explicitly grant these permissions as a precondition to installing the app. At runtime, the Android OS allows or denies use of specific resources based on the granted permissions (Section 2). In practice, this security model could use several improvements, e.g., informing users of the security implications of running an app, revoking/granting app permissions without reinstalling the app, or moving towards finer-grained permissions.

In fact, the Android permission model attracts emerging malware that challenges the system to exploit vulnerabilities in order to perform privilege escalation attacks—permission re-delegation attacks [7], confused deputy attacks, and colluding attacks [15]. As a result, users can have sensitive data leaked or subscription fees charged without their consent (e.g., by sending SMS messages to premium numbers via the SMS related Android permissions, as the well-known Android malwares Zsone and Geinimi do [18]). While most of these attacks are first initiated when a user downloads a third-party app to the device, to make matters worse, even stock Android devices with pre-installed apps are prone to exposing personal privacy information due to their higher privilege levels (e.g., the notorious HTCLogger app [5]).

Previous research efforts focus either on single-release permission characterization and effectiveness [6,9,13] or on other permission-related security issues [7,8,15,17]. Unfortunately, there have been no studies on how the Android permission system has evolved over the years, which could uncover important security artifacts beneficial to improving the security of the ecosystem. We discuss previous work in Section 7.

In this paper, we study the evolution of the Android ecosystem to understand whether the permission model is allowing the platform and its apps to become more secure. Following a systematic approach, we use three different types of characterizations (third-party app permissions vs pre-installed app permissions, and two permission classifications from Google). We study multiple Android platform releases over three years, from *Cupcake* (April 2009) to *Ice Cream Sandwich* (December 2011). We use a stable dataset of 237 evolving third-party apps covering 1,703 versions (spanning

a minimum of three years). Finally, we investigate pre-installed apps from 69 firmwares, including 346 pre-installed apps covering 1,714 versions. To the best of our knowledge, this is the first longitudinal study on Android permissions and the first study that sheds light on the co-evolution of the whole Android ecosystem: platform, third-party apps, and pre-installed apps.

Our overall conclusion is that the security and privacy of the ecosystem (platform and apps) do not improve, at least from the user's point of view. For example, the evolution moves more and more toward violating the *principle of least privilege*, a fundamental security tenet. Specifically, our study of the permission evolution of the Android ecosystem leads to the following observations:

1. **The number of permissions defined in Android platform tends to increase, and the `Dangerous`-level set of permissions is the most frequent and continues to grow.** There were 103 Android permissions in the first widely-used release (API level 3); the number of permissions has grown to 165 in the most current release (API level 15). Furthermore, the `Dangerous`-level permissions is always the largest group across all API levels, e.g., 60 out of 165 permissions in API level 15, and is still growing.
2. **Added platform permissions cater to hardware manufacturers and their apps, rather than third-party developers.** Nearly half (49.1% in API level 15) of all permissions are not accessible to third-party developers. Furthermore, of all the *added* permissions between API levels 3 to 15, most (49 out of 62) are in privilege levels that are not available to third-party developers, e.g., `Signature` and `signatureOrSystem` levels.
3. **Android platform permissions are not becoming more fine-grained.** After carefully examining the Android permissions from API level 3 to 15, we observed that most permission changes are *not* geared towards fine-grained instances of previous permissions. In other words, the platform does not seem to be moving towards more fine-grained permissions, which would in general be a step towards increased privacy or security. Instead, the permission changes indicate clearly that the Android platform is striving to give more flexibility and control to smartphone vendors, e.g., HTC, Motorola, Samsung, by providing them with permissions of higher privilege.
4. **Permission additions dominate the evolution of third-party apps, of which `Dangerous` permissions tend to account for most of the changes.** From the analysis of third-party apps, we found that the number of occurrences of adding Android permissions is significantly higher than the number of deleted permissions. Surprisingly, permission changes are not due to changes in the platform. Interestingly, among those additions, newer versions of apps tend to favor adding `Dangerous` permissions most often (66.11% of permission increases in apps consisted of at least one more `Dangerous` permission).
5. **Macroscopic and microscopic patterns emerge when studying evolution of permission usage.** We found evidence that `Dangerous` permission usage sometimes oscillate as an application evolves, which might indicate that developers are unclear about certain permission definitions, and their correct usage.
6. **An increasing number of apps are violating the principle of least privilege.** The tendency of developers to request permissions that their apps do not need causes an app to become overprivileged (as is the case for 44.8% of apps).
7. **The power and privilege of pre-installed apps is growing.** Sixty-six percent of pre-installed apps are overprivileged. Furthermore, pre-installed apps have more power to control and customize Android devices through Android platform-defined and self-defined higher protection level permissions, e.g., `Signature`- and `SignatureOrSystem`-level permissions. Though granting vendors higher privilege is not surprising, end-users (the actual owners of the device) still have security concerns [5, 11]. We argue that since pre-installed apps have greater power over the device, the developers of pre-installed apps must understand and accept their responsibility to protect the end user.

Implications and Suggestions. Our work leads to the following recommendations for increasing the security and privacy of Android users.

1. **Securing the ecosystem must start at the Android platform.** The trends we reveal in the evolution of the Android platform conjure up many security and privacy concerns. The security of the Android ecosystem could improve dramatically by focusing on improving the security of the Android platform by: (a) cautiously increasing the set of `Dangerous`-level permissions, (b) balancing the security of users and convenience of vendors, and (c) offering fine-grained permissions to app developers.
2. **App certification should enforce checks against over-privileged requests.** The existence of over-privileged apps, which are increasing in number, is an indication of, at best, carelessness, and, at worst, greed or malice of the app developer. Checks should be incoporated to discourage permission over-privilege.
3. **App permission evolution and fluctuation indicate developer confusion in selecting legitimate permissions.** Indicated by not only the macro- and micro-evolution patterns of permissions, but also by the tendency of apps to become overprivileged, the the struggling battle of developers to select a set of legitimate permissions for their Android apps is clearly shown in our work.More emphasis should be put on correct permission usage to aid developers in selecting the approriate permissions to use.
4. **Pre-installed manufacturer apps need to be subject to far more scrutiny, as they could be the weakest link.** Pre-installed apps have significant power: (a) they do not require user approval for installation, as they come with the device, (b) they can usually not be removed, even if the user tries to, (c) they get access to higher-privileged permissions, and (d) they are often overprivileged. Pre-installed apps, with all their power, could cause significant damage to the device and user if compromised, thus pre-installed developers must be held to a higher security standard than normal developers.

2. THE ANDROID PLATFORM BASICS

We now proceed to present an overview of the Android platform, Android permission model and a set of definitions for the concepts we use throughout the paper.

2.1 Android Platform

Android was launched as an open-source mobile platform in 2008 and is widely used by smartphone manufacturers, e.g., HTC, Motorola, Samsung. The software stack consists of a custom Linux system, the Dalvik Virtual Machine (VM), and apps running on top of the VM. Each app runs in its own copy of the VM with a different user id, hence apps are protected from each other. A permission model, explained shortly, protects sensitive resources, e.g., the hardware and stored data. In this model, resources are protected by permissions, and only apps holding the permission (which is granted when the app is installed) are given access to the permission-protected resource.

API Levels. To facilitate app construction, the Android platform provides a rich framework to app developers. The framework consists of Android packages and classes, attributes for declaring and accessing resources, a set of Intents, and a set of permissions that applications can request. This framework is accessible to apps via the Android application programming interface (API). The Android platform has undergone many changes since its inception in 2008, and each major release forms a new *API level*. In this paper we studied all major API levels, from level 3 (April 2009) to level 15 (December 2011); levels 1 and 2 did not see wide adoption. With each API upgrade, the older replaced parts are deprecated instead of being removed, so that existing applications can still use them [4].

2.2 Android Apps

In addition to the platform, the Android ecosystem contains two main app categories: third-party and pre-installed.

Third-party apps are available for download from Google Play (previously known as Android Market [2]) and other app stores, such as Amazon [11]. These Android apps are developed by individual third-party developers, which can include software companies or individuals around the world. Malicious apps, designed for nefarious purposes, form a special class of third-party apps.

Pre-installed apps come along with the devices from the vendors. They are developed and loaded in the devices before the devices ever reach the user in the market. These apps can be designed and configured exclusively per device model depending on the needs of particular manufacturers and phone service carriers by the vendor developers.

We studied permission evolution and usage in all components of the ecosystem: platform, third-party apps and pre-installed apps.

2.3 Android Permissions

The set of all Android permissions is defined in the `AndroidManifest.xml` source file of the Android platfrom [10]. To access resources from Android devices, each Android app, third-party and pre-installed alike, requests permissions for resources by listing the permissions in the app's `AndroidManifest.xml` file. When the user wants to install an app, this list of permissions is presented and confirmation is requested; if the user confirms the access, the app will have the requested permissions at all times (until the app is uninstalled). The latest platform release, API Level 15, contains a list of 165 permissions; examples of permissions are `INTERNET` which allows the app to use the Internet, `ACCESS_FINE_LOCATION` which gives an app access to the GPS location, and `NFC` which lets the app use near-field communication. Android defines two categories of Android permissions: *Protection Level* and *Functionality Group*, described next.

API level	Android platform	SDK codename	Total permissions	Release (mm-dd-yy)
15	4.0.3	Ice Cream Sandwich MR1	165	12-16-11
14	4.0.2 4.0.1	Ice Cream Sandwich	162	11-28-11 10-19-11
10	2.3.4 2.3.3	Gingerbread MR1	137	04-28-11 02-09-11
9	2.3.2 2.3.1 2.3	Gingerbread	137	12-06-10
8	2.2.x	Froyo	134	05-20-10
7	2.1.x	Eclair MR1	122	01-12-10
6	2.0.1	Eclair 0 1	122	12-03-09
5	2.0	Eclair	122	10-26-09
4	1.6	Donut	106	09-15-09
3	1.5	Cupcake	103	04-30-09

Table 1: Official releases of the Android platform; base and tablet versions are excluded.

Protection Level. The levels refer to the intended use of a permission, as well as the consequences of using the permission.

1. `Normal` permissions present minimal risk to Android apps and will be granted automatically by the Android platform without user's explicit approval.

2. `Dangerous` permissions provide access to the user's personal sensitive data and various device features. Apps requesting dangerous permissions can only be installed if the user approves the permission request. These are the *only* permissions displayed to the user upon installation.

3. `Signature` permissions signify the highest privilege; they can only be obtained if the requesting app is signed with the device manufacturer's certificate.

4. `signatureOrSystem` permissions are only granted to apps that are in the Android system image or are signed with the same certificate in the system image. Permissions in this category are used for certain special situations where multiple vendors have applications built into a system image and need to share specific features explicitly because they are being built together.

Note that the definition of protection level clearly constrains the privilege for each Android permission: third-party apps can only use `Normal` and `Dangerous` permissions. However, pre-installed apps can use permissions in all four protection levels. When third-party apps request `Signature` or `SignatureOrSystem` permissions, the request is ignored by the platform.

Functionality categories. Android also defines a set of permission categories based on functionality; in total there are 11 categories, with self-explanatory names: `Cost Money`, `Message`, `Personal Info`, `Location`, `Network`, `Accounts`, `Hardware Controls`, `Phone Calls`, `Storage`, `System Tools` and `Development Tools`. There is also a `Default` category that is used when no category is specified in the definition of an Android permission [3].

3. DATASET DESCRIPTION

In this section, we describe the process we used to collect the permission datasets from the Android ecosystem.

3.1 Platform Permissions Dataset

Table 1 presents the evolution of the platform permissions: for each API level (column 1) we show the platform release number (column 2), the textual codename of the release (column 3), the number of permissions defined in that release (column 4), and the release date (last column). Note that we exclude API levels 1 and 2, as the platform only gained wide adoption starting with API level 3. Also, we exclude releases 3.x (named Honeycomb, API levels 11–13); Honeycomb can be regarded as a separate evolutionary branch as it was designed for tablets only, not for smartphones, its source code was not open-source at release, and it was eventually merged into platform version 4.0.

To obtain the permission definitions for each API level, we extracted the file `AndroidManifest.xml` from each release [10]. We then analyzed the changes in permissions between successive releases.

3.2 Apps Permissions Dataset

Third-party apps. We characterize permission usage evolution in third-party apps based on a *stable set* of 237 popular apps with 1,703 versions that span at least three years. We chose these apps because they are widely-used, have releases associated in each API level, and have more than one release per year; hence we could observe how apps evolve and how changes in the platform might lead to changes in apps.

Selecting this stable dataset was far from trivial, and was an involved process. First, we seeded the dataset with 1,100 apps (Top-50 free apps from each category) [16]. Then we crawled historic versions of apps from online repositories, and then retrieved their latest versions from Google Play [1,2]; in total, this initial set contained 1,420 apps with 4,857 versions. Next, we selected only those apps that had at least one version each year between 2009 and 2012. Finally, after eliminating those apps that did not match our requirements, we obtained the stable dataset of 237 apps with 1,703 versions, with each app's evolution spanning at least three years.

Pre-installed apps. Pre-installed apps are much more difficult to obtain because they are not distributed online by vendors—they come with the phone; moreover, the sets of pre-installed apps vary widely among phones and manufacturers. Therefore, to collect pre-installed apps, we used a different process compared to third-party apps. First, we gathered the firmwares of multiple phone vendors—HTC, Motorola, Samsung, and LG—from various online sources. Next, we unpacked the firmwares and extracted the pre-installed apps inside. In total, we collected 69 firmwares over the years which contained 346 pre-installed apps with 1,714 versions.

Permission collection. To obtain the permission list for each app, we use the tool `aapt` on each app version to extract the `AndroidManifest.xml` file, which contains the permissions requested by that version [10]. After obtaining the set of manifest files, we parse the manifest files to get the full list of the permissions used by each app version.

Our analysis is based on these datasets. The datasets contain applications from a large number of developers across a broad range of categories. Thus, we believe that our datasets reflect Android app permission variation and evolution in a meaningful way.

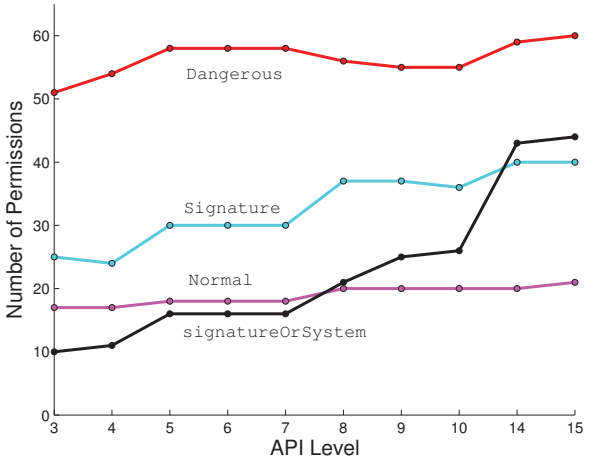


Figure 1: Protection Levels, e.g. Normal, Dangerous, Signature, signatureOrSystem, evolving over API levels.

4. PLATFORM PERMISSION EVOLUTION

We study the evolution of the Android platform permissions through a fine-grained, qualitative and quantitative analysis of permission changes between API levels. As we discussed in Section 2, the Android platform defines the list of all permissions in the framework's source code file `AndroidManifest.xml` for each API level. Since the API level directly reflects what permissions Android platform offers, we use the API level as the defining indicator to compare the Android permission changes.

4.1 The List of Permissions is Growing

As shown in Table 1, the number of Android permissions in each API level is significantly increasing. In early 2009, API level 3 had 103 Android permissions, while there are now 165 Android permissions in API level 15. The net gain of 62 permissions was the result of adding 68 new permissions and removing 6 existing ones. We present the permission evolution by protection level and functionality category.

In Figure 1, we show the permission evolution by protection levels (the levels were described in Section 2). We observe that the number of permissions in each protection level is increasing. In addition, we find that most of the increased permissions across different API levels belong to the protection levels `Signature` and `signatureOrSystem`, which indicates that most of the introduced Android permissions are only accessible to vendors, e.g., HTC, Motorola, Samsung, and LG. This raises significant security concerns for at least two reasons: (1) users have no control over the pre-installed apps, as the apps are already present when the phone is purchased, and (2) a flaw in a pre-installed app will affect all phones whose firmware contained that app. To illustrate the danger associated with pre-installed apps, consider the notorious `HTCLogger` pre-installed app, in which users of certain HTC phones were exposed to a significant security flaw. `HTCLogger` was designed to log device information for the development community in order to debug device-specific issues; as such, the app collects account names, call

API level	Dev tools	Sys tools	Accounts	Cost Money	Hardware Controls	Location	Messages	Network	Personal Info	Phone calls	Storage	Default
3	36	35	1	2	6	4	5	5	6	3		
4	-1	+2,-2				+1					+1	+2
5		+3	+4						+2			+7
6												
7												
8		+7										+6, -1
9					+1			+2	-2			+2
10												
14		+2		+1	+2,-1		+1	+1	+5	+1	+1	+12
15		+1						+1				+1
Overall	-1	+13	+4	+1	+2	+1	+1	+4	+5	+1	+2	+29

Table 2: Permission changes per API level and permission categories.

Dangerous permission	Category
READ_HISTORY_BOOKMARKS	Personal Info
WRITE_HISTORY_BOOKMARKS	Personal Info
READ_USER_DICTIONARY	Personal Info
READ_PROFILE	Personal Info
WRITE_PROFILE	Personal Info
READ_SOCIAL_STREAM	Personal Info
WRITE_SOCIAL_STREAM	Personal Info
WRITE_EXTERNAL_STORAGE	Storage
AUTHENTICATE_ACCOUNTS	Accounts
MANAGE_ACCOUNTS	Accounts
USE_CREDENTIALS	Accounts
NFC	Network
USE_SIP	Network
CHANGE_WIFI_MULTICAST_STATE	System Tools
CHANGE_WIMAX_STATE	System Tools

Table 3: Added `Dangerous` permissions and their categories.

and SMS data, GPS location, etc. Unfortunately, the app stored the collected information without encrypting it and made it available to any application that had the Internet permission [5].

In Table 2, we show the permission evolution by functionality categories: each column contains a category, each row corresponds to an API level, and cell data indicates the number of permissions added and deleted in that API level; note that, the first row shows the number of permissions in each category of API 3. We find that the number of permissions in nearly all the categories is increasing, with the exception of the `Personal Information` category, which yielded a decrease in the number of permissions from API 8 to 9, as shown in Table 2. After grouping the Android permissions into the 11 functionality categories, we find that the `Default`, `System_Tools` and `Development_Tools` categories contribute to most of the increases. Newly-added permissions in these categories allow developers and applications to take advantage of the evolving hardware capabilities and features of the device. We now proceed to providing observations on permission evolution at a finer-grained level.

4.2 Dangerous Group is Largest and Growing

From Figure 1, we can see that the `Dangerous` permission level (the levels were introduced in Section 2.3) vastly outnumbers all other permission types at all times. Note that the `Dangerous` permission set is still growing, even though it is already the largest. We further investigated the growth of permissions in the `Dangerous` protection level.

As shown in Table 3, `Dangerous` permissions are added in 5 out of 11 categories. Most of them are from personal data-related categories, e.g, `PERSONAL_INFO`, `STORAGE` and `ACCOUNTS`. We believe that this evolutionary trend shows that the Android platform provides more channels to harvest personal information from the device, which could increase the privacy breach risk if these permissions may be abused by Android apps.

4.3 Why are Permissions Added or Deleted?

To understand the rationale behind permission addition and deletion, we studied the commit history (log messages and source code diffs) of the Android developer code repository [10].

We found that, in most cases, permissions are added and deleted to offer access to more functionality offered by the device. Advances in the hardware strongly motivate such permission evolution. For instance, in API level 9, new hardware technology for near-field communication led to the introduction of a permission to access NFC. In API level 15, a permission to access WiMAX is introduced in order to access 4G networks.

Permissions can also be deleted to accommodate new smartphone features when they are removed and replaced by new permissions. For example, `READ_OWNER_DATA` was deleted after API level 8, but two new, related permissions, `READ_PROFILE` and `READ_SOCIAL_STREAM` were added in level 14.

Interestingly, some permissions were added in the earlier API levels while deleted later, as the associated functionalities are made available to public without manifest-declared permissions. For example, `BACKUP_DATA` was added in API level 5, but deleted in level 8, because the backup/restore function was made available to all apps by default.

Furthermore, most of the added permissions are permissions categorized as `Default`, `System_Tools` and `Development_Tools`, which are mostly used to access system level information to function and debug the Android apps. However, as we discussed before, most of those permissions are in the `Signature` and `signatureOrSystem` protection levels that are only available to vendor developers in pre-installed apps. This indicates that the added permissions facilitate the development of pre-installed apps by vendor developers, instead of third-party apps by third-party developers. The extended aid to vendors is somewhat adverse, since third-party developers are the dominant and active force in the Android ecosystem.

4.4 No Tendency Toward Finer-grained Permissions

Finer-grained permissions in Android, e.g., separating the advertisement code permissions from host app permissions [14], have been advocated by security groups from both academia

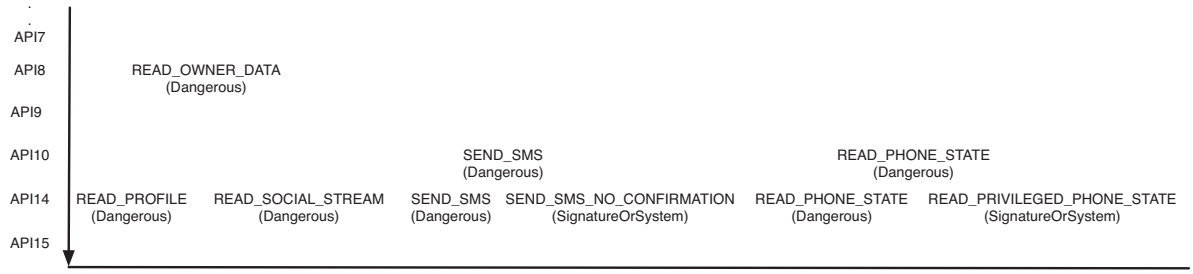


Figure 2: Functionally-similar permissions added and deleted between API levels.

and industry [9, 12, 16]. The basis for finer-grained permissions is the *principle of least privilege*, i.e., giving apps the minimum number of permissions necessary to provide a certain level of service.

We investigated whether Android permissions are becoming more fine-grained over time. After carefully examining the Android permissions from API level 3 to 15, we observe that the permission changes do *not* tend towards becoming more fine-grained. We found only one possible example of a permission splitting in `READ_OWNER_DATA`. However, there is no indication that the two new permissions were specifically designed to replace the previous one, as shown in the first example of Figure 2. Overwhelmingly, the permission changes indicate that the Android platform is giving more flexibility and control to the phone vendors. For example, as shown in Figure 2, `SEND_SMS` and `PHONE_STATE` permissions exist in both API level 10 and 14, but the newly added Android permissions `SEND_SMS_NO_CONFIRMATION` and `READ_PRIVILEGED_PHONE_STATE` gives the app a higher privileged access to the device. Further, those higher privileged permissions are `signatureOrSystem` permissions, which can only used by vendor developers. In summary, we do not observe the evolution of Android permissions that is trending to provide more fine-grained permissions.

5. THIRD-PARTY APPS

We now change our focus and investigate the variation and evolution of permissions from the perspective of the driving force of the Android ecosystem: the apps. We investigate two types of apps, *third-party apps* and *pre-installed apps*; we present and discuss the permission usage of Android apps across different versions and their evolution.

5.1 Permission Additions Dominate

We analyzed the permissions added and deleted in the 1,703 versions of the 237 third-party apps in our stable dataset. In Figure 3(a) we show the distribution of permission changes; on the x-axis we show the number of permission changes: permission additions are marked positive, permission deletions are marked negative. Note that the bulk of the changes are to the right of the origin (0 changes means no permission change), we can conclude that most apps add permissions over time, with some apps adding more than 15 permissions. Only a small number of apps, about 10, delete permissions, and the deletions are limited to at most 3 permissions.

We present the total numbers of permission addition and deletion events in the stable dataset in Table 4: column 2 illustrates that the addition of permissions occurs much more frequently than the deletion of permissions. To disambiguate between genuine permission additions and additions induced by changes in the platform (e.g., as a result of added functionality), we also computed the permission changes induced by changes in the Android platform, which we show in column 3 of Table 4). Surprisingly, these induced changes only account for a small number of the permission changes: less than 3% of either additions or deletions. In sum, we were able to conclude that permission changes, which consist mostly of additions, are not due to changes in the platform.

	Total changes	**Induced by platform changes**
Add	857	14 (1.63%)
Delete	183	5 (2.73%)
Total	*1040*	*19 (1.82%)*

Table 4: App permission changes in the stable dataset.

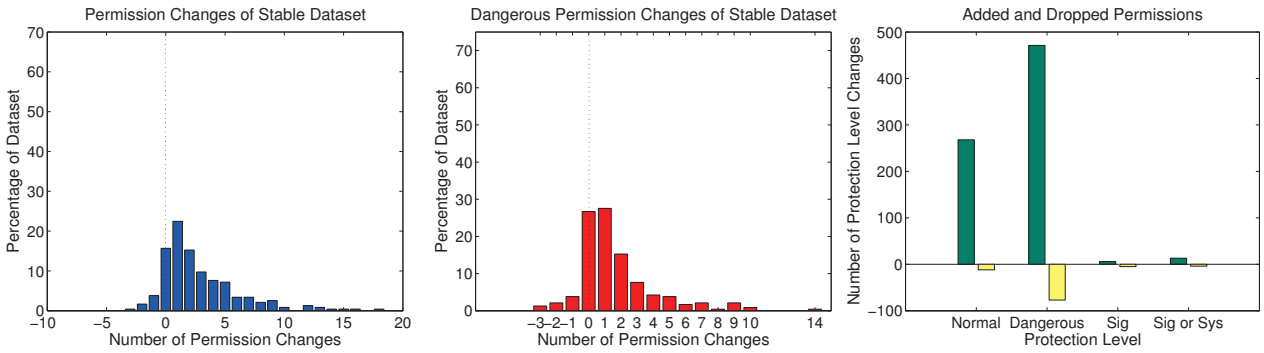
Android permission	**In Top 20?**
ACCESS_NETWORK_STATE	✓
WRITE_EXTERNAL_STORAGE	✓
WAKE_LOCK	✓
GET_ACCOUNTS	×
VIBRATE	✓

Table 5: Most frequently added permissions in the stable dataset.

We now set out to answer the question: *what is the primary cause for the permission additions*? We show the Top-5 most frequently added and dropped permission in the first column of Table 5 and Table 6; column 2 of these tables will be explained shortly. For the added permissions, we found that Android apps became more aggressive in asking for resources, by asking for new permissions. For instance, the Android apps adopt permissions such as `WAKE_LOCK`, `GET_ACCOUNTS`, and `VIBRATE`. `WAKE_LOCK` prevents the processor from sleeping or the screen from dimming, hence allowing the app to run constantly without bothering the user for wake-up actions. `VIBRATE` enables the phone to vibrate for notifying the user when the corresponding apps invokes some functionality. In order to meet the increasing requirement of storage, `WRITE_EXTERNAL_STORAGE` is added to enable writing data into the external storage of the device such as an SD card. We note that permissions that do not improve the user experience, e.g., `ACCESS_MOCK_LOCATION` and `INSTALL_PACKAGES`, the apps simply drop them.

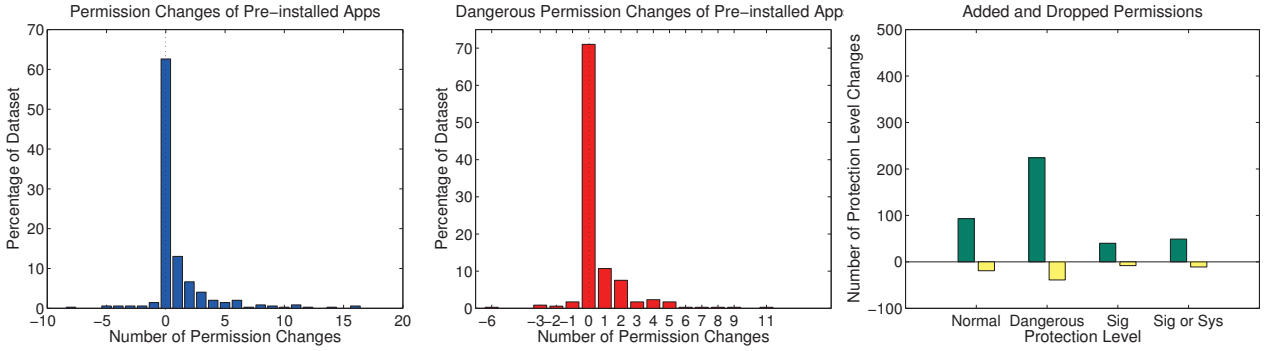
As Android Apps are increasingly adding new permissions, users are naturally have security and privacy concerns, e.g., *how can they be sure that apps do not abuse permissions?*

For comparison, in Table 7, we list the Top-20 permissions that Android malwares request (and abuse), as reported by Zhou and Jiang [18]. We now come back to column 2 in Tables 5 and 6; the columns show the result of comparing the added (and respectively, deleted) permissions in our stable dataset with the Top-20 malware permission list. A '✓' means the corresponding Android permission is in the Top-



(a) (b) (c)

Figure 3: Permission and protection level changes in the third-party apps.



(a) (b) (c)

Figure 4: Permission and protection level changes in the pre-installed apps.

Android Permission	In Top 20?
ACCESS_MOCK_LOCATION	×
READ_OWNER_DATA	×
INSTALL_PACKAGES	×
RECEIVE_MMS	×
MASTER_CLEAR	×

Table 6: Most frequently deleted permissions in the stable dataset.

20 malware permission list, while a '×' means the permission is not in the list. We found that most of the added permissions are on the malware list, while none of the dropped permissions are on the list. Though we certainly can not claim these third-party apps are malicious, the trend should concern users: as apps gain more powerful access, the overall system becomes less secure. For example, in the *confused deputy* attack, a malicious app could compromise and leverage a benign app to achieve its malevolent goals [15].

5.2 Apps Want More Dangerous Permissions

We now proceed to investigate the added permissions in the `Dangerous` protection level as they introduce more risks.

Figure 3(b) shows that 66.11% of permission increases in apps required at least one more `Dangerous` permission. In more detail, we list the frequently used `Dangerous` permissions in the first column of Table 8. We found that `WRITE_EXTERNAL_STORAGE` is the most requested `Dangerous` permission, in which sensitive personal or enterprise files can be written to external media. This permission is also a hotspot for most malicious activities. `INTERNET`, `READ_PHONE_STATE`, and `WAKE_LOCK` are also requested frequently by the new versions of the apps. The first two are needed to allow for embedded advertising libraries (ads), but these third-party ads are also raising privacy concerns of abusing the user's personal information. We then cross-checked this list with the Top-20 malware permissions [18], as shown in column 2 of Table 8. We observed that 9 of the 16 frequent permissions listed are also frequently used by malicious apps. This significant overlap intensifies our privacy and security concerns.

Permission	% of apps using it
INTERNET	97.8%
READ_PHONE_STATE	93.6%
ACCESS_NETWORK_STATE	81.2%
WRITE_EXTERNAL_STORAGE	67.2%
ACCESS_WIFI_STATE	63.8%
READ_SMS	62.7%
RECEIVE_BOOT_COMPLETED	54.6%
WRITE_SMS	52.2%
SEND_SMS	43.9%
VIBRATE	38.3%
ACCESS_COARSE_LOCATION	38.1%
READ_CONTACTS	36.3%
ACCESS_FINE_LOCATION	34.3%
WAKE_LOCK	33.7%
CALL_PHONE	33.7%
CHANGE_WIFI_STATE	31.6%
WRITE_CONTACTS	29.7%
WRITE_APN_SETTINGS	27.7%
RESTART_PACKAGES	26.4%

Table 7: Top-20 most frequent permissions requested by malware (from Zhou and Jiang [18]).

5.3 Macro and Micro Evolution Patterns

The characterization of permission changes we provided so far, in terms of absolute numbers (added/deleted), reveals

Dangerous permission	In Top 20?
WRITE_EXTERNAL_STORAGE	✓
WAKE_LOCK	✓
READ_PHONE_STATE	✓
ACCESS_COARSE_LOCATION	✓
CAMERA	×
INTERNET	✓
ACCESS_FINE_LOCATION	✓
READ_LOGS	×
READ_CONTACTS	✓
RECORD_AUDIO	×
BLUETOOTH	×
CALL_PHONE	✓
CHANGE_WIFI_STATE	✓
GET_TASKS	×
MODIFY_AUDIO_SETTINGS	×
MANAGE_ACCOUNTS	×

Table 8: Frequently used `Dangerous` Android permissions of stable dataset.

Macro pattern	Frequency
0→1	90.46%
1→0	8.59%
1→0→1	0.84%
1→0→1→0	0.11%

Table 9: Macro evolution patterns of permission usage in the stable dataset.

the general trend toward apps requiring more and more permissions. In addition, we also performed an in-depth study where we looked for a finer-grained characterization of permissions evolution in terms of "patterns", e.g., repeated occurrences of permission changes.

Macro patterns. To construct the macro patterns, we use 0→1 and 1→0 as the basic modes, where '0' represents the state that the corresponding app does not use a particular permission, '1' represents the state that the corresponding app uses a particular permission, and '→' represents a state transition. In Table 9, we tabulate the macro-patterns we observed in the stable dataset, along with their frequencies. We found that the permission additions dominate the permission changes (0→1 has a 90.46% frequency), as pointed out earlier in Section 5.1. We also found occurrences of other interesting patterns, e.g., permissions being deleted and then added back, though these instances are much less frequent.

Micro patterns. Some `Dangerous` permissions appear to be confusing developers. For example, the location permissions `ACCESS_COARSE_LOCATION` and `ACCESS_FINE_LOCATION`, provide different levels of location accuracy, on GSM/WiFi position and GPS location, respectively. Location tracking has been heavily debated because it could possibly be used to violate the user's privacy. We found that app developers handled the adding and deleting of these `Dangerous` location permission in an interesting way; to reveal the underlying evolution patterns of used by the `Dangerous` location permissions, we have done a case study of micro-patterns on two widely used location permissions, `ACCESS_COARSE_LOCATION` and `ACCESS_FINE_LOCATION`. We found that, although the most frequent macro evolution pattern of location permission is 0→1, the micro evolution patterns of the location permissions are quite diverse.

In Table 10, we tabulate the micro-patterns we observed for the location permission alone. For instance, 0→Both→Fine means both location permissions are used at first, then the `ACCESS_COARSE_LOCATION` permission is deleted in a later version of the app. 0→Fine→0→Fine shows the app added `ACCESS_FINE_LOCATION` at first, dropped it in a subsequent version, and finally, added back again. Though the table indicates several micro-patterns, note that using both location permissions dominates, with 50% of the total, which shows that more and more apps tend to include both location permissions for location tracking. We are able to make two observations. First, evolution patterns requesting `Dangerous` permissions clearly show the struggling balance between app usability and user privacy during the evolution of apps. Second, the patterns reveal that developers of third-party apps may be unclear with the correct usages of the `Dangerous` location permissions, which highlights the importance for the platform to be more clear on how to properly handle `Dangerous` permissions.

Micro pattern	Frequency
Both	6.67%
Fine→Both	10.00%
Fine→Coarse	3.33%
Coarse→Both	10.00%
0→Both	20.00%
0→Fine	10.00%
0→Coarse	26.70%
0→Fine→Both	3.33%
0→Both→Fine	3.33%
0→Both→Coarse	3.33%
0→Fine→0→Fine	3.31%

Table 10: Micro evolution patterns for the location permissions; Fine represents the `ACCESS_FINE_LOCATION` permission, Coarse represents the `ACCESS_COARSE_LOCATION` permission, and Both means both Fine and Coarse are used.

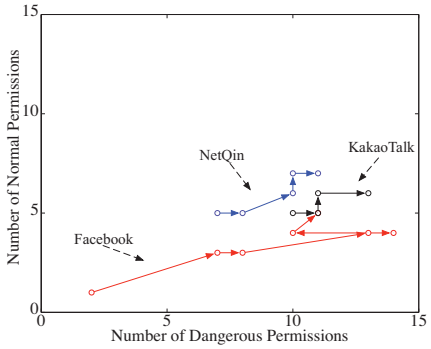


Figure 5: Permission trajectories for popular apps.

Permission trajectories. Due to the observed diverse permission evolution patterns, we plot the number of `Normal` against `Dangerous` permissions to visualize trajectories as apps evolve. We found many interesting trajectories, and highlight three, e.g., Facebook (red), KakaoTalk (black) and NetQin (blue), in Figure 5. Facebook added `Dangerous` permissions in great numbers early on, but recently they have removed many and instead added more slowly. Both NetQin and KakaoTalk continue to add permissions from either one permission level or both permission levels with each new version that is released. These diverse trajectories of popular apps again highlight the need for the the platform to provide

Micro pattern	Frequency
Legitimate →Over	58.57%
Over→Legitimate	32.14%
Over→Legitimate→Over	7.86%
Over→Legitimate→Over→Legitimate	0.71%
Over→Legitimate→Over→Legitimate→Over	0.71%

Table 11: Evolution patterns of the privilege levels of the stable dataset, where Legitimate **represents legitimate privilege and** Over **represents overprivilege.**

Permission	Protection level
GET_TASKS	Dangerous
MODIFY_AUDIO_SETTINGS	Dangerous
WAKE_LOCK	Dangerous
NFC	Dangerous
GET_ACCOUNTS	Normal

Table 12: Most added permissions from the Legitimate→Over (58.57%) subset of apps.

better references of Android permissions to developers.

5.4 Apps Are Becoming Overprivileged

Extra permission usage may lead to overprivilege, a situation in which an app requests the permission, but never uses the resource granted. This could increase vulnerabilities in the app and raise concern of security risks. In this section, we investigate the privilege patterns to determine whether Android apps became overprivileged during their evolution.

To detect overprivilege, we ran the Stowaway [8] tool on the stable dataset (1,703 app versions). As shown in Figure 7, we found that 19.6% of the newer versions of apps became overprivileged as they added permissions, and 25.2% of apps were initially overprivileged and stayed that way during their evolution. Although the overall tendency is towards overprivilege, we could not ignore the fact that 11.6% of apps decreased from overprivileged to legitimate privilege, a positive effort to balance usability and privacy concerns.

In addition, similar to the evolution patterns of permission usage, we also study the evolution patterns of overprivilege status for each app; we present the results in Table 11. We found that the patterns Legitimate→Over and Over→Legitimate dominate at 58.57% and 32.14%, respectively. However, like in the patterns of permission usage, we also found other diverse patterns during the evolution of apps, which again shows that there may be confusion for third-party developers when deciding on what permissions to use for their app.

In Table 12 and 13, we further refine the observations to show the kinds of permissions involved in the dominating patterns: we observe that `Dangerous` permissions are the major source that causes an app to be overprivileged, which again emphasizes that developers should exercise more care when requesting `Dangerous` permissions.

6. PRE-INSTALLED APPS

Pre-installed apps have access to a richer set of higher-privileged permissions, e.g., at the `Signature` and `signatureOrSystem` levels, compared to third-party apps, which gives pre-installed apps access to more personal information on the device [11]. Thus, we should investigate how Android permissions are used in pre-installed apps. We conducted a permission-change analysis for pre-installed apps

Permission	Protection level
READ_PHONE_STATE	Dangerous
ACCESS_COARSE_LOCATION	Dangerous
WRITE_EXTERNAL_STORAGE	Dangerous
ACCESS_MOCK_LOCATION	Dangerous
VIBRATE	Normal

Table 13: Most dropped permissions from the Over→Legitimate (32.14%) subset of apps.

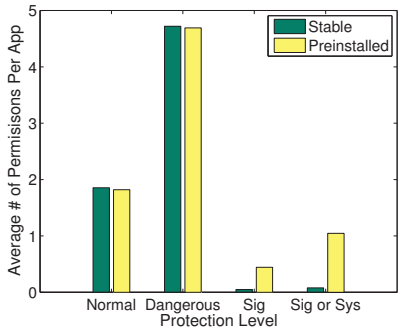


Figure 6: Average number of permissions per app, for each protection level, from stable and pre-installed datasets.

in a manner similar to the stable dataset. We present the results in Figure 4. Figures 4(a) and 4(b) indicate that permission usage is relatively constant, e.g., 62.61% of pre-installed apps do not change their permissions at all, which is significant when compared to our third-party apps with only 15.68%. Further, from Figure 4(c) and 6, pre-installed apps request many more `Signature` and `signatureOrSystem` level permissions than third-party apps, while at the same time requesting nearly just as many `Normal` and `Dangerous` level permissions. This shows that pre-installed apps have a much higher capability to penetrate the smartphone. Interestingly, the vendors also have the ability to define their own permissions inside the platform when they customize the Android platform for their devices. For example, HTC defines its own app update permission, `HTC_APP_UPDATE`.

The power of pre-installed apps requires great responsibility by vendors to ensure that this power is not abused. On one hand, vendors are able to customize pre-installed apps to take full advantage of all the hardware capabilities of the device, as well as create a brand-personalized look-and-feel to enhance user experience. On the other hand, users cannot opt out of pre-installed apps, and in most cases, cannot uninstall the pre-installed apps, which raises the question: *why should users be forced to trust pre-installed apps?* Hindering that trust is our finding that, despite being developed by vendors, 66.1% of pre-installed apps were overprivileged.

What if the power of pre-installed apps is used against the user with malicious intent? For example, the marred pre-installed app `HTCLogger` and other reported security compromised apps have already indicated such security risks do exist and can significantly damage the smartphone and/or the user data [5, 11]. The vendors' `Signature` and `signatureOrSystem` level permissions can be exploited by malicious apps to do an array of damaging actions, such as wiping out user data, sending out SMS messages to premium numbers, recording user conversations, or obtaining the device location data of the device [11].

As we analyzed the evolution of Android platform permis-

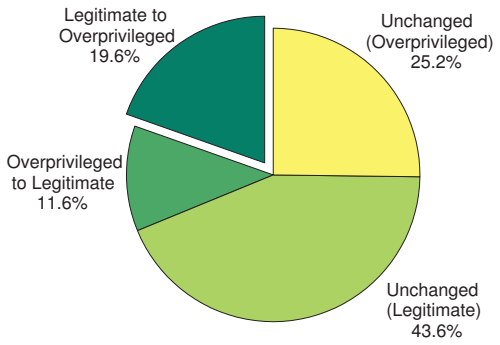


Figure 7: Overprivilege status and evolution in the stable dataset.

sions, it was interesting to see the evolution trends benefit vendors, rather than users. With the power vendors have in pre-installed apps, developers of pre-installed apps should be more careful in their development as they represent the trusted computing base (TCB) of the Android ecosystem. Up until now, there has not been any clear regulations or boundary definitions that protect the user from pre-installed apps. We argue that, since pre-installed apps have more power and privilege over Android devices, vendors need to realize their responsibility to protect the end-user.

7. RELATED WORK

None of the prior works on Android permissions has focused on understanding how Android permissions and their use evolve in the Android ecosystem.

Android permission characterization and effectiveness. Barrera et al. [9] introduced a self-organizing method to visualize permissions usage in different app categories. A comprehensive usability study of Android permissions was conducted through surveys in order to investigate Android permissions' effectiveness at warning users, which showed that current Android permission warnings do not help most users make correct security decisions [6]. Chia et al. [13] focused on the effectiveness of user-consent permission systems in Facebook, Chrome, and Android apps; they have shown that app ratings were not a reliable indicator of privacy risks.

Permission-related Android security. Enck et al. [17] presented a framework that read the declared permissions of an application at install time and compared it against a set of security rules to detect potentially malicious applications. Ongtang et al. [12] described a fine-grained Android permission model for protecting applications by expressing permission statements in more detail. Felt et al. [8] examined the mapping between Android API's and permissions and proposed Stowaway, a static analysis tool to detect over-privilege in Android apps. Permission re-delegation attacks were shown to perform privileged tasks with the help of an app with permissions [7]. Grace et al. [11] used Woodpecker to examined how the Android permission-based security model is enforced in pre-installed apps and stock smartphones. Capability leaks were found that could be exploited by malicious activities. DroidRanger was proposed to detect malicious apps in official and alternative markets [19]. Zhou et al. characterized a large set of Android malwares, e.g., accumulating fees on the devices by subscribing to premium services by abusing SMS_related Android permissions [18]. An effective framework was developed to defend against privilege-escalation attacks on devices [15].

8. CONCLUSION

We have investigated how Android permission and their use evolve in the Android ecosystem via a rigorous study on the evolution of the platform, third-party apps, and pre-installed apps. We found that the ecosystem is becoming less secure and offer our recommendations on how to remedy this situation. We believe that our study is beneficial to researchers, developers, and users, and that our results have the potential to improve the state of practice in Android security.

Acknowledgements

This work was supported in part by National Science Foundation award CNS-1064646, by a Google Research Award, by ARL CTA W911NF-09-2-0053, and by DARPA SMISC Program W911NF-12-C-0028.

9. REFERENCES

[1] Freewarelovers, May 2012. http://www.freewarelovers.com/android.
[2] Google Play. https://play.google.com/store, May 2012.
[3] Android. Android-defined Permission Category. http://developer.android.com/reference/android/Manifest.permission_group.html, May 2012.
[4] Android Developer. Android API. http://developer.android.com/guide/appendix/api-levels.html, May 2012.
[5] Android Police. Massive Security Vulnerability In HTC Android Devices. http://www.androidpolice.com/2011/10/01/massive-security-vulnerability-in-htc-android-devices, October 2011.
[6] A.P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin and D. Wagner. Android Permissions: User Attention, Comprehension, and Behavior . In *SOUPS*, 2012.
[7] A.P. Felt, H. Wang, A. Moshchuk, S. Hanna and E. Chin. Permission Re-Delegation: Attacks and Defenses. In *USENIX Security Symposium*, 2011.
[8] A.P.Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android Permissions Demystified. In *ACM CCS*, 2011.
[9] D. Barrera, H.G. Kayacik, P.C. van Oorschot and A. Somayaji. A Methodology for Empirical Analysis of Permission-based Security Models and its Application to Android. In *ACM CCS*, 2010.
[10] Google. Android Open Source Project, May 2012. http://source.android.com/.
[11] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic Detection of Capability Leaks in Stock Android Smartphones . In *NDSS*, 2012.
[12] M. Ongtang, S. McLaughlin, W. Enck and P. McDaniel. Semantically Rich Application-Centric Security in Android. In *ACSAC*, 2009.
[13] P. H. Chia, Y. Yamamoto, and N. Asokan. Is this App Safe? A Large Scale Study on Application Permissions and Risk Signals . In *WWW*, 2012.
[14] P. Pearce, A.P. Felt, G. Nunez and D. Wagner. AdDroid: Privilege Separation for Applications and Advertisers in Android . In *ACM AsiaCCS*, 2012.
[15] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A. Sadeghi, and B.Shastry. Towards Taming Privilege-Escalation Attacks on Android . In *NDSS*, 2012.
[16] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A Study of Android Application Security. In *USENIX Security Symposium*, 2011.
[17] W. Enck, M. Ongtang and P. McDaniel. On Lightweight Mobile Phone Application Certification. In *ACM CCS*, 2009.
[18] Y. Zhou and X. Jiang. Dissecting Android Malware: Characterization and Evolution. In *IEEE S &P*, 2012.
[19] Y. Zhou, Z. Wang, Wu Zhou and X. Jiang. Hey, You, Get off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets . In *NDSS*, 2012.

Practicality of Accelerometer Side Channels on Smartphones

Adam J. Aviv, Benjamin Sapp, Matt Blaze and Jonathan M. Smith
University of Pennsylvania
{aviv,bensapp,blaze,jms}@cis.uepnn.edu

ABSTRACT

Modern smartphones are equipped with a plethora of sensors that enable a wide range of interactions, but some of these sensors can be employed as a side channel to surreptitiously learn about user input. In this paper, we show that the accelerometer sensor can *also* be employed as a high-bandwidth side channel; particularly, we demonstrate how to use the accelerometer sensor to learn user tap- and gesture-based input as required to unlock smartphones using a PIN/password or Android's graphical password pattern. Using data collected from a diverse group of 24 users in controlled (while sitting) and uncontrolled (while walking) settings, we develop sample rate independent features for accelerometer readings based on signal processing and polynomial fitting techniques. In controlled settings, our prediction model can on average classify the PIN entered 43% of the time and pattern 73% of the time within 5 attempts when selecting from a test set of 50 PINs and 50 patterns. In uncontrolled settings, while users are walking, our model can still classify 20% of the PINs and 40% of the patterns within 5 attempts. We additionally explore the possibility of constructing an accelerometer-reading-to-input dictionary and find that such dictionaries would be greatly challenged by movement-noise and cross-user training.

Categories and Subject Descriptors

I.5 [**Pattern Recognition**]: Applications

General Terms

Security, Design, Experimentation, Measurement, Performance

Keywords

Smartphone Security, Accelerometer, Side Channels

1. INTRODUCTION

Smartphone motion sensors measure the movement and orientation of the phone in space, and sensors have been used in a wide variety of tasks, notably in gaming applications. Applications are generally granted access to these sensors without much concern and without notifying the user; however, certain sensors may be able to measure much more than just the user's intention within a single application.

It has recently been shown that the gyroscopic motion sensor, which measures the smartphone's orientation (*e.g.*, pitch or roll), is capable of inferring where on a touchscreen a user taps/touches [6, 34]. Such inferences constitute a side channel, potentially conveying secure input intended for a foreground application to a background application that has access to the sensor. This new class of smartphone side channels are a direct result of the new computer interaction layer promoted by smartphones. As compared to traditional computer platforms, smartphones are tactile, hand-held devices, and users provide input by physically touching and gesturing on the touchscreen. These actions implicitly shift and adjust the device in measurable (and machine predictable) ways.

In this paper, we continue this line of investigation into *sensor-based side channels* by focusing on the smartphone's accelerometer sensor's capability in this domain. The key question we investigate is: Considering a background application with access to the accelerometer, *what can the background application learn about user input to the foreground application via the accelerometer readings?* We show that the accelerometer is sensitive to user input and can function as a side channel, and in applicable comparisons, we found that accelerometer based techniques perform nearly as well, or better, than gyroscopic based techniques.

We focus on inferring two common smartphone secure input types using the accelerometer sensor: four-digit PINs (tap/touching) and the Android password pattern (gesturing/swiping). We collected accelerometer readings from 24 users, 12 entering PINs and 12 entering patterns. Using standard machine learning techniques, we show that accelerometer measurements reliably identify the PIN or pattern that was entered. In our experiments, when selecting from a uniform test set of 50 possible PINs or patterns, our models can predict the PIN entered 43% and pattern 73% of the time within 5 guesses. Further, when we introduce movement noise caused by users walking while providing input, our models can still predict PINs 20% of the time and patterns 40% of the time within 5 guesses. We also employ a Hidden Markov Model (HMM) to predict variable-length sequences of digits pressed in a PIN or swipes in a password pattern. On this considerably harder sequence prediction problem (where the random chance of being correct is roughly 0.01%), we can predict PINs 40% of the time and patterns 26% of the time within 20 guesses

To summarize, this paper makes the following contributions:

- We perform a large user study of sensor-based side channels (24 users and over 9,600 samples); the first study to consider both controlled (users sitting) and uncontrolled settings (users walking).

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

- We develop novel machine learning features for accelerometer readings that are sample-rate independent and based on signal processing and polynomial fitting techniques.
- We demonstrate that the accelerometer sensor is a highly capable side channel against security-sensitive input, such as PINs and password patterns, and general input based on touch/tapping or gesture/swiping. In comparisons to previous results, where applicable, accelerometer data performs nearly as well, or better, than gyroscopic data.
- We observe that there is reasonable consistency across users and devices; however, movement noise and user variance may be too great to construct an accelerometer-reading to input dictionary mapping.

Finally, based on these results, and previous sensor-based side channel results [6, 7, 22, 24, 34], it is clear that the security model for smartphones with respect to on-board sensors should be reconsidered. In this paper, we also propose *context-based* [23, 9, 4] sensor access revocation policy for smartphones, such that applications with access to sensors either block (or fail) when attempting to read from such sensors while sensitive input is being provided.

2. RELATED WORK

Gyroscopic Smartphone Side Channels. Cai *et al.* first proposed using on-board smartphone sensors as a side channel to learn users' input [6]. Their system, *touchlogger*, describes a side channel that employs the gyroscopic orientation sensor to determine broadly where a user touches on a large keypad. Their results were very encouraging, and in controlled settings, were able to infer which of the 10 regions a user touched with 70% accuracy. Similarly, in *taplogger*, Xu *et al.* refined the techniques for inferring user input from gyroscopic data [34][1], and were able to predict PIN-like input based on a telephone key pad. Xu *et al.*'s models detected all the digits of the PIN within three inference steps; that is, upon the successive, non-overlapping predictions for each digit, all digits of the PIN were *covered*. However, Xu *et al.* does not detail a process for choosing a permutation of the predicted labels. For example, after three predictions, there are three possible values for each digit in a four-digit PIN, thus requiring, in the worst case, 81 possible guesses to predict the input. Surprisingly, Xu *et al.* does not apply standard sequence prediction techniques, such as Hidden Markov Models (HMM), to link each individual prediction together.

In work parallel to our own, Milluzo *et al.* developed TapPrints [22] which uses a combination of gyroscopic and accelerometer data to infer tap events and location of tap events on tablet and smartphone keyboards. Additionally, in parallel, Cai *et al.* developed further techniques using both the accelerometer and gyroscope to infer numeric and soft-keyboard input on tablets and smartphones [7].

Our work differs from these previous and parallel techniques in that we investigate using *only* the accelerometer sensor to infer user input. Additionally, we demonstrate that input based on swipe gesturing as well as input based taps/touches are susceptible to sensor-based side channels. We also explore the use of new sample-rate independent features, and finally, we investigate the effects of motion-noise, such as a user walking, which can have a considerable effect on the accuracy of motion-based inference techniques.

Accelerometer Smartphone Side Channels. *ACCessory* [24] by Owusu *et al.* is closer to our work. In *ACCessory*, the authors demonstrate that the accelerometer can be used as a basic side channel to infer short sequences of touches on a soft keyboard, and that standard machine learning techniques can be employed to infer input like passwords. Similarly, we show that the accelerometer can be used to infer secure input, and we also demonstrate that input can be classified with a sequence predictor.

Our work differs from Owusu *et al.* in that we also demonstrate that swiping can be inferred from accelerometer data in addition to touch input. We additionally show that certain touch input, like PIN entry, can be classified at a much higher rate and with fewer guesses than suggested by Owusu *et al.*. *ACCessory* was able to classify input strings of length 6 with 60% accuracy, but needed 2^{12} guesses to achieve that result. In a similar experiment with PIN entry, we showed that the PIN entered can be classified with 40% accuracy within 20 guesses on average (see Figure 15).

In interesting related work, Marquardt *et al.* showed that smartphone accelerometers can infer more than input occurring on the phone. They developed *(sp)iphone* that collected accelerometer readings while the smartphone is placed next to a keyboard [20]. The vibrations of a user typing on the keyboard is recorded by the phone and generally interpreted to predict what was typed on the keyboard. This technique is similar to acoustic keyboard side-channels that use audio recordings to surreptitiously learn user input [1, 35], as well as keystroke timing techniques [31].

Smartphone Side Channels. Side channels against secure smartphone input have been previously demonstrated for the Android password pattern input. In earlier work, we described *smudge attacks* that are based on observing the oily residues remaining on touchscreens after a pattern is entered [2]. The side channel described here has a similar goal, but is based on internal sensors rather than external observations. An additional observation made in [2] is that the Android password pattern is more susceptible to the side-channel than other secure input types, such as PINs or text-based passwords. Our conclusion is that inferring password patterns using the accelerometer is *generally* more effective than inferring PINs, but in specific situations such as sequence prediction, PINs can be slightly easier to infer.

Other sensors and recording devices have been proposed as side channels on smartphones. Shlegel *et al.* proposed Soundcomber [30] and demonstrated that a malicious app that has access to the microphone can learn the difference between general chatter and tone dialing, effectively learning the numbers a user calls. Similarly, Xu *et al.* considered information that can be leaked if a malicious app has access to the smartphone's camera [33], and Cai *et al.* investigate sniffing sensors including the microphone, camera, and GPS receiver [8].

3. BACKGROUND

PINs. Both Apple iOS and Android based smartphones support PINs as a screen lock mechanism. PINs are the primary iOS screen lock interface, but Android provides two other options: a graphical password pattern (see below) or a pass-phrase consisting of both numbers and letters. A PIN consists of a sequence of four digits, 0-9, and digits may repeat. Thus, there are a total of 10,000 possible PINs, and iOS will lock down the phone after 10 failed attempts, while Android allows for 20 failed attempts. In addition to securing the device, PINs are also used in banking applications, particularly Google Wallet [13] requires a user to enter a PIN to confirm transactions.

Password Pattern. The Android password pattern is a graphical password scheme that requires users to enter a sequence of swipes that connect contact points in a three-by-three grid. The user must maintain contact with the screen while entering a pattern, and a

[1] Xu *et al.* do investigate accelerometer data in *taplogger* for purposes other than inferring the location of tap events on the screen.

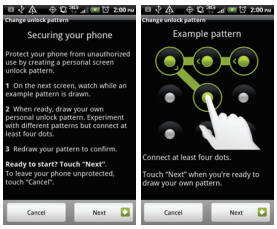


Figure 1: Android Password Pattern Instructions

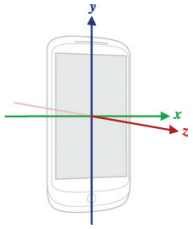


Figure 2: Accelerometer Axis of Measurement (Source [10])

user's pattern must minimally contact four points (see Figure 1). Android allows for 20 failed pattern entry attempts before locking the device permanently. Despite its seeming complexity, only 389,112 possible patterns exist [2], and likely, many of those patterns are completely unusable for general daily use: in our experience (see Section 5), using a randomly chosen pattern as a security credential will be too difficult to enter reliably. The number of actual *human-usable* patterns remains an interesting question; we hypothesize that it is at least an order of magnitude less than the total of available patterns.

Accelerometer Sensor. The accelerometer sensor measures linear movements in three dimensions, side-to-side, forward-and-back, and up-and-down (labeled x, y, and z respectively in Figure 2). Upon each reading, a data element is provided that contains the acceleration reading in all three linear directions, and the units are in m/s^2 with the force of gravity considered. Note that the accelerometer sensor measures different movement than the gyroscopic sensor, which senses the orientation of the phone, *i.e.*, the pitch and roll angles. Although certain movements can be measured in both, *e.g.*, tilting the phone forward and back, others are only measured by one sensor or the other, *e.g.*, holding the phone face up and moving it left would only be measured by the accelerometer sensor.

Accelerometers have been previously studied in the computer science community, and researchers have shown that accelerometer readings can provide a rich source of information about the actions of individuals [3, 18, 21, 28, 29]. Using accelerometers as a user interface (UI) enhancement has also been proposed [18, 19, 27]. The accelerometer sensor is used in many applications, for example in the *Bump* application [32], an application to quickly exchange contact information by "bumping" smartphones together. More light weight applications also make use of the accelerometer, for example applications that simulate a "light saber" use the accelerometer to determine when to play a sound effect [14].

4. ATTACK SCENARIO

We consider an attacker who wishes to learn the secure input of smartphone users via an accelerometer side channel. An attacker may gain access to accelerometer data in a wide variety of ways – *e.g.*, the attacker finds a phone where an application has written accelerometer data to the device storage. We consider a more active attacker who distributes a malicious smartphone application that can run in the background, has access to the accelerometer, and can communicate over the network. As an example of the kinds of input an attacker may be able to learn, we focus on the information that is leaked by two common input types, entering a PIN or Android password pattern that is used to lock the smartphone.

To this end, the malicious application is aware when the phone initially wakes and, thus, the smartphone will prompt a user for a PIN or password pattern while the malicious application is running in the background. The application then activates the accelerometer sensor, recording measurements for a short time period. We found that it takes 2.4 seconds to enter a pattern and 1.3 seconds to enter a PIN, on average, so the accelerometer does not need to be active for very long. The accelerometer measurements are eventually sent over the network to be analyzed offline.

The attacker's goal at this point is to develop a method for comparing the captured accelerometer data to a corpus of labeled accelerometer data[2]. That is, the attacker has at his/her disposal accelerometer data that he/she knows was collected when a particular PIN or pattern is entered. The problem of identifying the PIN or pattern entered reduces to a classic machine learning problem: Given previously labeled input, what is the label of the unknown input? In this scenario, the label is the PIN or pattern of the victim.

We consider two scenarios in our experiments for the attacker's capabilities to make this comparison to the corpus at his/her disposal. In the first scenario, we assume that the attacker has a large corpus, and samples of the PIN or pattern he/she is trying to learn can be found in the corpus. In the second scenario, we assume that the attacker does not have samples in the corpus, or not enough to generate a strong model. Instead the attacker has a limited set of labeled samples of individual swipes or touch events, such as a swipe from left to right on the screen or the touch of a particular digit.

In our experiments, we model these two scenarios by first considering a sample set of 50 patterns and 50 PINs. Here the goal of the experiment is to measure how accurately a pattern and PIN can be identified based on previously seen input. In the second scenario, where the attacker does not have sufficient labeled data, the goal of the experiment is to measure the accuracy of a sequence predictor that tries to identify a pattern by making a sequence of smaller predictions (*e.g.*, a single swipe or digit press). We present more details of our machine learning setup in Section 6.

Of course, an important question is: What can an attacker do with the information learned? Clearly, if the attacker has learned a user's password pattern, it is only useful if the attacker gains physical access to the victim's phone at some later point because the Android password pattern is not a widely used security mechanism. Granted, this is a reasonable attack scenario. However, learning a user's smartphone unlock PIN may be applicable in other settings if the user reuses his/her PIN, such as an ATM PIN or in an online banking application [5].

More broadly, we focus on PINs and Android password patterns because they represent a larger set of user input on touchscreens that is composed of point touching and gesturing. Demonstrating an accelerometer side channel against these input types is an example of a broader family of sensitive touchscreen inputs that may be susceptible to this side channel.

5. DATA COLLECTION

We built two applications to model the attacker's perspective and determine if a background application with access to the accelerometer can infer input to the foreground one. The first application prompts users to enter a PIN, and records accelerometer data in the background; similarly, the other application prompts the user to enter a pattern while recording accelerometer data in the background. A visual of the applications can be found in Figure 3.

We recruited 24 users to participate in the core study: 12 users entered password patterns, and 12 users entered PINs. The users in our experiment were surprisingly diverse. Two users were left

[2]The attacker could build such a corpus by distributing an application that requires users to enter patterns for other purposes, such as [11, 15, 26].

Model Name	Chipset	Pattern/PIN	Sample Rate
Nexus One	Snapdragon S1	5/5	$\sim$ 25 Hz
G2	Snapdragon S2	6/6	$\sim$ 62 Hz
Nexus S	Hummingbird	1/0	$\sim$ 50 Hz
Droid Incredible	Snapdragon S1	0/1	$\sim$ 50 Hz

Table 1: Android smartphones used in experiments, their chipsets, number times used in either pattern or PIN experiments, and their observed accelerometer sample rate.

handed, and less than 50% of the users owned a smartphone. All users, however, have used a smartphone at some point. Only two users locked their phone, and they did so using a PIN and not a password pattern.

We used a total of four phones in our experiment, two were provided by us: Nexus One and G2. If the user owned an Android phone, we installed the application directly on his/her phone for the experiment. This occurred twice, and experiments were also conducted on a Nexus S and Droid Incredible. All the phones in our experiments indicate through the standard API that the accelerometer can sample at 76 Hz. In practice, we observed this to almost never be the case, and even phones with the same chipset sampled at different rates. This is likely due to slight differences in the Android OS installed. Details about the phones used in the experiments can be found in Table 1.

Experiment Overview. The experiment for both PINs and patterns consisted of two rounds. In the first, the users were asked to sit at a table and enter in 50 PINs/patterns in random order using their dominant hand a total of 5 times. Following, we asked users to walk in a circle (around our lab) while entering in the same set of 50 PINs/patterns using their dominant hand. We provided very little oversight during the experiment: After providing instructions, we periodically checked in on users' status, but did not provide further instruction.

For each user, we collected 5 samples of each PIN/pattern in a controlled setting (*i.e.*, sitting) and 1 sample in an uncontrolled setting (*i.e.*, walking). We considered the sitting data set as training data, and the walking data set as the testing data, only testing against it once all the models were tuned using the sitting data. All the results presented, unless otherwise noted, are an average across multiple runs of a 5-fold cross validation using the training set, while the user was sitting.

It is important to note that the patterns and PINs used in the experiment are not the users' real patterns or PINs, and that real-world users will likely be very well practiced at entering in their own PINs or patterns. This familiarity could affect the way (*e.g.*, the way the phone moves in space) a user enters a pattern or PIN. We do not model this in our experiments (indeed, performing such an experiment on users' actual secure input could be seen as unethical). However, our test users, by the end of data collection, have entered each PIN and pattern a number of times, and many even commented about their familiarity with the patterns/PINs in the test set upon completion.

PIN Data. PINs were selected at random. A total of 50 PINs were used in the experiment, and all twelve users entered the same set of PINs a total of 5 times. We only considered accelerometer data when the user entered the PIN correctly, and users are re-prompted until the PIN was entered correctly. In addition to recording accelerometer readings, we also log the timing of the touch events to ensure that the accelerometer data matches the timing of PIN entry. We considered all accelerometer readings that occurred within 50 ms of entering the first digit and 50 ms after entering the last digit.

Pattern Data. Pattern data is collected in a similar way to PIN data – twelve users enter a set of 50 patterns a total of 5 times and

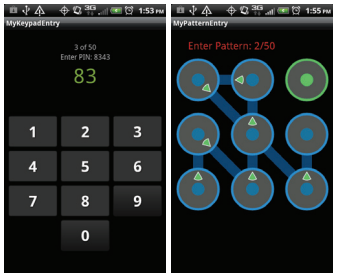


Figure 3: PIN and Pattern Entry Applications

touch information is logged when a user gestures across a contact point. We initially selected a set of 50 patterns at random. However, we quickly discovered that the vast majority of the patterns selected were surprisingly hard to enter. The patterns were convoluted and overly complicated, and in a initial test of the application, our test users reported that it took many iterations (5+) to enter the pattern correctly. As a result, we wished to use a set of reasonable and representative password patterns that our test users could reliably enter on their first attempt. We developed two simple criteria to select patterns at random that meet this requirement.

The first criterion limits the number of cross-overs, that is, it limits the number of swipe segments that cross (or double back) over previous swipe segments (*e.g.*, the pattern in Figure 3 contains a single cross-over). The motivation for this criterion is that users would likely move in consistent directions. We anticipate that users would generally select the next contact point in region near the current contact point. The second criterion restricts contact points that are untouched, requiring that untouched contact points be generally near other untouched contact points. Similar to the cross-over criteria, this restriction again assumes that users will likely connect points in nearby regions.

We do not argue that real world users apply these criteria while selecting their patterns, but in our experience, these criteria do produce patterns that our test users found reasonable to enter. Studying user selection criteria for password patterns is beyond the scope of this paper, and we are unaware of any such study.

6. ANALYSIS AND ML TECHNIQUES

In this section, we present our analysis of the collected accelerometer data as well as present our machine learning techniques for classifying data. The accelerometer measurements for both PINs and patterns consist of a sequence of readings in each accelerometer dimension (x,y,z). In addition to the accelerometer measurements, we also record the timing of touch events. A touch event for a PIN is when the user presses a digit, and a touch event for a pattern is when a user swipes across a contact point. The touch events are used to properly align the accelerometer data.

A malicious application distributed by an attacker will not have direct access to touch events from other applications—if it did, then there would be no need to employ side channels. A malicious application must also determine when secure input begins and how to segment the accelerometer readings. Automatically detecting touch events from raw accelerometer data is beyond the scope of this study; however, other machine learning techniques (or information from other side channels) could be employed to solve this problem. Additionally, techniques suggested in [34] could be applied here, but in our investigation, we found that it may be ineffective with low sample rates and gentler tap events, as what seems to occur for single hand input. Further, the techniques in [34] would be ineffective for gesture input, as required to determine touch events for patterns.

Feature	Length	Description
STATS	6	Root mean square, mean, standard deviation, variance, max and min
3D-Poly-Deg	4	Parameters of a degree-3 polynomial fit
3D-Poly-STATS	6	STATS for a degree-3 polynomial fit reconstruction
iFFT-Poly	35	The inverse Discrete Fourier Transform (DFT) of a DFT of the 3-D polynomial fit curve using 35 samples.
iFFT-Acc	35	The inverse DFT of the DFT of the accelerometer readings using 35 samples.

Table 2: Features Set: Each feature is extracted in each linear direction in the accelerometer reading.

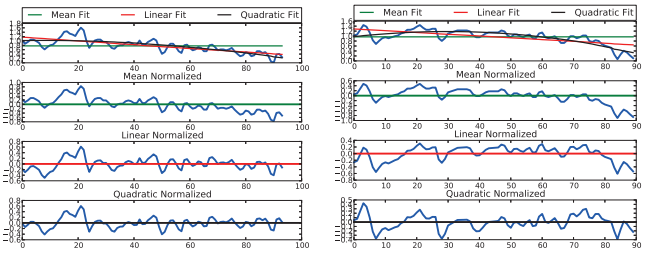


Figure 4: Visual example of normalization: In the top plot, the raw accelerometer data is presented with the appropriate mean, linear, and quadratic fits, and following plots show the affect on the raw accelerometer data when normalized to those fits, respectively.

6.1 Feature Extraction

In this section, we describe the feature set used as input to the machine learning classifiers. For notation, consider a stream of accelerometer readings $A = \{a_1, \ldots, a_n\}$ of size n. Each data value $a_i \in A$ contains four sub-values (or elements): a_i^x, the acceleration in the x direction; a_i^y, the acceleration in the y direction; a_i^z, the acceleration in the z direction; and, a_i^t, the time stamp of this reading. Additionally, allow A^d to refer to the projection of the d^{th} element of the readings in A, that is, $A^d = \{a_1^d, \ldots, a_n^d\}$.

As is, the accelerometer data is varied, affected by subtle tilts and shifts. For example, often the z dimension is close to 9.8 m/s^2, *i.e.*, the force of gravity. The first step in feature extraction is to normalize the readings in each dimension such that they fluctuate about 0. We use three normalized forms of A for feature extraction:

1. **Mean Normalization**: For each linear direction d, compute the mean $m^d = \text{mean}(A^d)$, and return: $A_m = \{a_i^d - m^d\}$.

2. **Linear Normalization**: Perform a linear fit and compute the fit curves $L^d = \{l_1^d, \ldots, l_n^d\}$ for each accelerometer direction d, and return: $A_l = \{a_i^d - l_i^d\}$.

3. **Quadratic Normalization**: Perform a quadratic fit and compute the fit curves $Q^d = \{q_1^d, \ldots, q_n^d\}$ for each accelerometer direction d, and return: $A_q = \{a_i^d - q_i^d\}$.

A visual example of the normalization is provided in Figure 4. Following the normalization, we have three representations of A, A_m, A_l, and A_q. Now, for each normalized accelerometer data stream, we extract the features in Table 2.

The first set of features extracted is standard statistics of the accelerometer stream (STATS), such as the root mean square, mean, standard deviation, variance, max and min. Each of these stats are computed for each normalization in each dimension, *e.g.*, for A_m, we compute STATS(A_m^x), STATS(A_m^y), STATS(A_m^z) and the resulting 18 features are appended to the feature vector.

The next two features are computed by first fitting a 3-degree polynomial to the accelerometer readings in each dimension. The

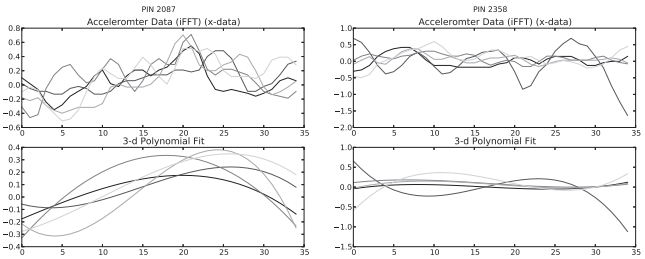


Figure 5: An example of polynomial fit features for PIN 2087 (*left*) and PIN 2358 (*right*). The top plot shows iFFT-ACC of the accelerometer data (just acceleration in the x dimension), and the bottom plot shows the 3-d polynomial fit (iFFT-Poly).

parameters of the fitted polynomial in each dimension are the next features added (3D-Poly-Deg); that is, d_3, d_2, d_1, d_0 from $f(t) = d_3t^3 + d_2t^2 + d_1t + d_0$ where t refers to the timestamp of the readings. Following, we compute the curve values at each time stamp in A^t and add the STATS of that curve as a set of features (3D-Poly-STATS).

The next two features, iFFT-Poly and iFFT-Acc, are sample-normalized forms of the polynomial curve and accelerometer stream. The goal is to use the consistency in the shape of the curves of both the polynomial fit and the accelerometer readings as features, but there is a large variance in the number of samples n across readings, even when a user enters the same PIN or pattern multiple times. We wish to instead use the curves as features in a sample-normalized way such that regardless of n, we can represent the stream in m values.

To solve this problem we use 1-dimensional Discrete Fourier Transforms (DFTs) with a resolution of $m = 35$ samples. More precisely, we compute

$$\text{real}(\mathcal{F}_m^{-1}(\mathcal{F}_m(A^d))).$$

This computation first encodes the signal using m complex frequency basis functions, then reconstructs the original signal from its compressed form. This preserves the general shape and values of the curve, but it normalizes the time domain to m samples and discards noisy high frequency components of the signal. We experimented with varied values of m and found that a small value of m did not preserve enough information, while a large value of m preserves too much variance because if $m > n$, the input is zero padded. We found that $m = 35$ to be a good compromise between these extremes, and it performed effectively for both PINs and patterns.

To further demonstrate this technique, in Figure 5 we visualize the iFFT-Acc and iFFT-Poly for accelerometer reading collected while a user entered in two different PINs (note, this is accelerometer readings in just the x dimension). Even though the same PIN was entered by the same user on the same smartphone, n varied between 59 and 112; however, you can see that regardless of the variance in n, there is a shared shape to the curves. This is what we wish to capture in our feature set.

In total, for each accelerometer reading, we use 774 features. That is, for each dimension (x, y, and z) and for each normalization, we extract 86 features, totaling $774 = 3 \times 3 \times 86$. In experiments, we found that all the features improve prediction results, and that these features were effective for both PINs and patterns, as well as single tap/touch and swipe/gesture events.

6.2 Machine Learning Classification

Two classification procedures are used in experimentation to match the attack scenario described in Section 4. Recall that we

wish to model two scenarios: (1) The attacker has a large corpus of labeled accelerometer data at his/her disposal and attempts to match unknown input to some label in the corpus; and (2), the unknown input is not in the corpus (or not well represented).

Logistic regression. To model the first scenario, where the attacker is matching unknown input to labels in a corpus, we train a multi-class logistic regression model on the feature vector labeled with the PIN or password pattern (we use the LIBLINEAR implementation [12]). For each possible label, the logistic regression finds a discriminating line in feature space to best separate examples of the label from examples of all other labels. Thus, the regression learns a weighted sum of the features described in Section 6.1 for each label.

Given accelerometer data from entering a PIN or pattern not used in training, the resulting logistic regression model will output a predicted label (*i.e.*, a PIN or pattern), or a set of labels ordered by the likelihood of being the true label. If the label matches the input, we consider this a successful prediction. We consider multiple guesses from the model as the ranking of the output label that matches the input label.

There are some limitations to this experiment because we only learn models for the known PINs and patterns in the training set; that is, the 50 pattern or 50 PINs used in the experiment as opposed to all 389,112 possible patterns and 10,000 possible PINs. However, picking from random chance of the possible 50 patterns would result in a 2% prediction accuracy. The model greatly exceeds random guessing by a factor of 20 or more for patterns and 9 or more for PINs.

Hidden Markov Models. To model the second scenario, where the attacker's corpus may not have sufficient samples of the unknown input, we build a classifier that can predict previously unseen sequences of patterns and PINs. To achieve this, we obtain the probability of each label from the output of the logistic regression classifiers, and use these as observation probabilities in a Hidden Markov Model (HMM). The HMM finds the most likely sequence of input patterns or PINs (*maximum a posteriori*) by jointly considering the probabilities of individual swipe or digit entry classifications along with the likely transitions between swipes or digit entries.

For example, for a four-digit PIN, the HMM jointly infers the most likely set of four digits given the individual beliefs in what digit was pressed at what time, and what digits are likely to follow other digits—certain combinations of digit transitions are impossible, and others are more likely than others. The same inference process can be used for patterns based on which swipes (connecting two contact points) are likely to follow previous swipes.

Formally, let ℓ_i be a possible label for position i in a sequence, and o_i its corresponding observed feature vector. Then, we obtain $p(\ell_i|o_i)$ from the logistic regression model for all ℓ_i—the probability that the label is ℓ_i given the data o_i. The transitions $p(\ell_{i+1}, \ell_i)$ are estimated via maximum likelihood from our training data; simply empirical averages of each transition in the training data. For a sequence of length k, the HMM determines the most probable joint assignment

$$(\ell_1^\star, \ldots, \ell_k^\star) = \arg\max_{(\ell_1,\ldots,\ell_k)} \prod_{i=1}^{k} p(\ell_i|o_i) \prod_{i=1}^{k-1} p(\ell_i, \ell_{i+1}).$$

Note that the joint space of possible labels $(\ell_1, \ldots, \ell_k)$ is combinatorial (exponential in k). Fortunately, efficient dynamic programming techniques exist to solve this exactly in $O(k^2)$ time.

In our experiments, we explore label spaces of different granularities. In an HMM over *unigrams*, each position in the sequence corresponds to a single swipe or digit. In an HMM over *bigrams* labels consist of a pair of swipes or consecutive digits. We quickly found that the unigrams performed poorly, and in the results below, we only use bigram HMMs. This is a proof of concept, and a larger model could incorporate even larger scope (larger grams), including refined transition matrices that account for human pattern/PIN selection factors.

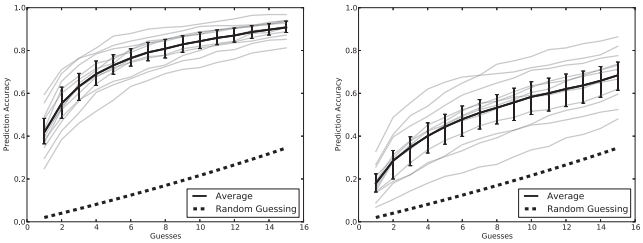


Figure 6: Prediction accuracy over multiple guesses for predicting patterns (*left*) and PINs (*right*). The shaded trend lines are individual users.

7. EVALUATION RESULTS

In this section, we present the results of our experiments for inferring PINs and patterns using accelerometer reading. We begin by modeling the first attacker scenario, where the attacker has access to a large corpus of labeled data. We additionally address trends in expanding the corpus from 50 PINs/patterns, and how such prediction models would fare. Next, we investigate a general prediction model based on Hidden Markov Models which addresses the second attacker scenario. All the results presented in this section, unless otherwise noted, are the average across five randomized runs of a five-fold cross validation.

7.1 PIN/Pattern Inference

To begin, we are interested in how distinguishable PIN/pattern inputs are based on accelerometer readings using the features described in Section 6. The data used in this experiment consists of the 50 PINs and 50 patterns collected from the 24 users while they were sitting. The experiment proceeds by performing a five-fold cross validation. Each of the five runs from a given user is randomly divided into five folds, and a model is constructed from the features extracted from four of the folds, and tested on the fifth. This process is repeated until all folds have been in the testing and training positions.

The results from this experiment are presented in Figure 6. The y-axis is prediction accuracy, and the x-axis is a of plot is the number of prediction or guesses attempted; that is, the logistic regression model output allows for a probabilistic ranking of the predicted labels based on how likely it is the true label. For example, two guesses refers to using the two top ranked predicted labels. If the true label is one of those two labels, we consider it accurately predicted with two guesses. The dark trend line refers to the average across all 12 users for PINs and 12 users for patterns. The error bars on this curve mark the 1st and 3rd quartiles. The grayscale lines are individual users, and the dotted line represents the prediction probability for random guessing[3]. We use this style in all graphs presented in this section unless otherwise noted.

Inspecting Figure 6, it is clear that accelerometer readings do leak sufficient information to differentiate between input of the same type. In all cases, across all users, our model can infer the precise PIN or pattern when selecting from the set of 50 PIN/patterns

[3]Note that the trend line for random guessing with multiple attempts is not linear because of conditional probabilities.

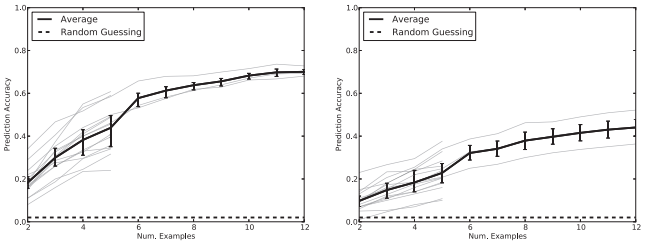


Figure 7: Trendline for how the number of examples affect prediciton for patterns (*left*) and PINs (*right*)). Note that we include an additional three users who provided 12 examples, and the original 24 users only provided 5 examples of each PIN/pattern.

at a rate substantially higher than random guessing. Upon the first prediction, for patterns, the model on average predicts with 40% accuracy, 20 times greater than random guessing of 2%; however, PIN inference only averages 18% across all users, just 9 times greater than random guessing. But, upon successive predictions, the models perform better: On the fifth prediction, the model can predict the pattern with 73% accuracy and PINs 43% of the time, a difference of $\sim$50% and $\sim$30% over random guessing, respectively. Considering prediction accuracy rates after multiple guesses is important because an attacker would likely have multiple attempts at guessing secure input, such as the 20 attempts provided by Android for unlocking the phone and the 10 attempts provided by iOS.

Example Trends. In the experiment above, each cross-validation uses just four examples for training while testing on the fifth. An interesting question is: *How would these models perform if more examples were available?* That is, we are interested in the example learning curve. To investigate this we recruited three additional users to enter in the same set of 50 patterns and 50 PINs a total of 12 times, each in the controlled, sitting endowment. while sitting using the same instructions as before. We then include those results with the original 24 users to see if there should be an increase in prediction accuracy with more training data.

To measure the effect of additional examples, we incrementally increase the number of examples (and folds) performed. Beginning with two examples for each PIN/pattern, we perform a two-fold cross validation. Following, we use three examples and perform a three-fold cross validation, and so on, until there are no more examples to include. The results of this experiment are presented in Figure 7: The x-axis is the number of examples used, and the y-axis is the prediction accuracy. For both patterns and PINs, there is a clear increase in inference accuracy as the number of examples increase. At the extreme, with 12 examples, patterns are inferred with an accuracy near 60% on the first prediction, and PINs are near 40%. Both PINs and patterns see diminishing returns on accuracy after 8-10 examples; the logarithmic growth of the learning curves is consistent with computational learning theory [16]. Overall, patterns, again, are more easily predicted via accelerometer data given the features we developed, plateauing at a prediction rate 50% greater than that of PINs.

Label Trends. Another important question is: *How would these models perform as the number of available labels increases?* That is, we are interested in the performance of a similar model that must predict from a set of 10,000 labels, rather than just 50, as would be the case if an attacker were targeting users generally. This scenario can be estimated by performing a sequence of five-fold cross validations, where in each step an additional label is included in training and testing. For example, in the first step, the model must select between two labels, and in the last step, it must select from 50, as before.

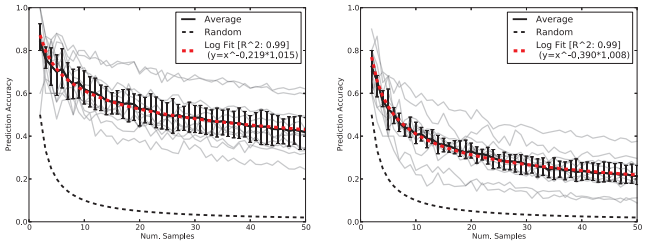


Figure 8: Trendline for the number of samples being selected from: patterns (*left*) and PINs (*right*)). Note that the accuracy rates closely match an inverse exponential.

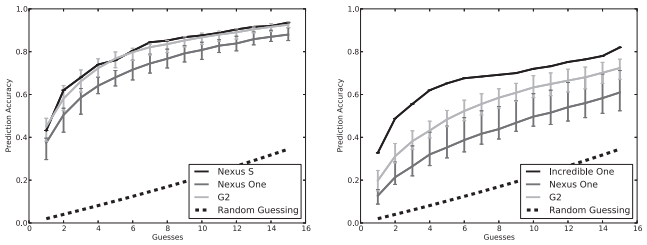


Figure 9: Prediction accuracy over multiple guesses for predicting patterns (*left*) and PINs (*right*) for different devices.

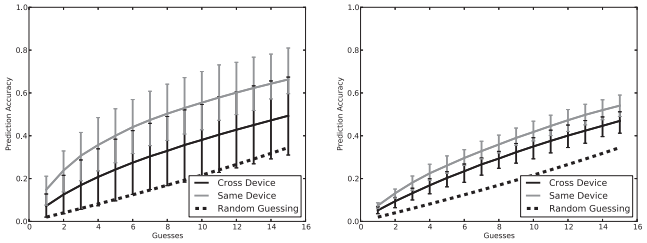


Figure 10: Prediction accuracy over multiple guesses for predicting patterns (*left*) and PINs (*right*) when training and testing on different devices.

The results of this experiment are presented in Figure 8: The x-axis is the number of included labels, the y-axis is the prediction accuracy, and the dotted line is the probability of random guessing. As the number of labels in the model increases, the average trend matches very closely ($R^2 > .99$) to an inverse exponential (in *dashed-red*), and using this trend line, we can extrapolate the performance of such a model (with 5 examples per label) predicting across any number of labels. For example, selecting from 10,000 PINs, we should expect an inference accuracy of about 2% on the first prediction, which is 277x greater or 8 orders of magnitude greater than random guessing. For patterns, if the model is selecting from 10,000 patterns, it should predict with an accuracy of 13% on the first prediction, and, if it was selected from all 389,112 possible patterns, it should predict with an accuracy of 6% on the first prediction, 23,567x greater or 14.5 orders of magnitude greater than random guessing. These are likely optimistic projections for our feature set, but these results do suggest that predicting input from a large label space using accelerometer readings is tractable, if an attacker were able to collect sufficient examples.

User and Device Effects. As noted in Table 1 and in Section 5, the data set contains rather large variance across devices and users. An important question is: *How does training on accelerometer readings from one device or user and testing on another device or user affect an attacker's inference capabilities?* Such results speak to the attacker's ability to construct a large and diverse corpus to use

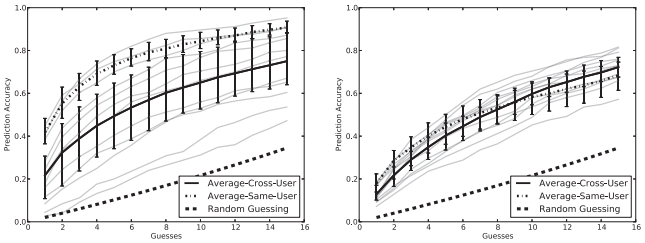


Figure 11: Prediction accuracy over multiple guesses for predicting patterns (*left*) and PINs (*right*) for training on 11 users and testing on one.

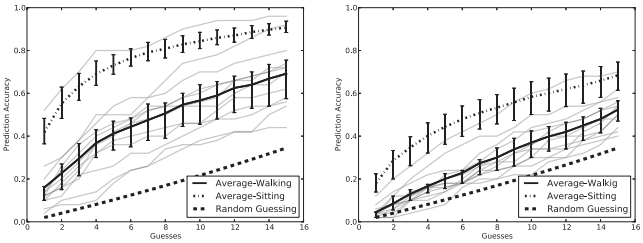


Figure 12: Prediction accuracy over multiple guesses for predicting patterns (*left*) and PINs (*right*) while the user is ***walking***. The shaded trend lines are individual users.

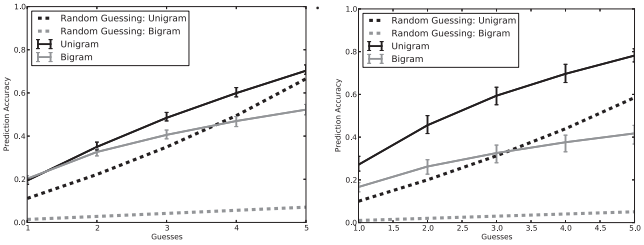


Figure 13: Prediction accuracy for uni- and bigrams for patterns (*left*) and PINs (*right*) with 5 guesses. Note that there are 9 and 10 possible unigrams and 72 and 100 possible bigrams for patterns and PINs, respectively.

in training on users/devices previously unseen.

To begin, we investigate prediction performance for training and testing on the same device for the same user. These results are presented in Figure 9. As we might expect, devices with higher accelerometer sample rates (refer to Table 1) tend to perform better; however, the decrease in performance for lower sample rates is not as extreme as was seen in [24]. For patterns, there is a small drop in inference performance between the Nexus S and the Nexus One, although the Nexus S effective sample rate is double that of the Nexus One. PINs seem more affected by sample rate issues, there is at least a 50% drop in performance between the highest sample rate device and the lowest sample rate one. Yet, all devices perform well above random guessing, suggesting that the features are reasonably resilient to sample rate fluctuations, as addressed by the sample-normalized features (see Section 6).

However, in order to show that the attacker can construct a comprehensive dictionary, we must show that training and testing on different devices and different users is also effective. In Figures 10 and 11, we present the results of experiments to test such a capability. First, in Figure 10 we present two trend lines: one where training and testing occurred on the same device and one where training and testing occur on different devices. As expected, training and testing on different devices performed worse than using the same device. This decline was fairly significant for patterns; however, the decline was relatively small for PINs by comparison.

In Figure 11, we present the results of experiments where we constructed a model trained on all but one user in the data set, and tested on the remaining. This experiment most closely resembles the scenario of an attacker with a large corpus trained on varied users and devices. Interestingly, although patterns are inferred at a reasonable rate on average, there is great variance. Inspecting the gray-scale lines for individual users, some users perform fractions better than random guessing, while others perform as well or better than testing and training on the same user (the dash-dotted trend line). PINs, surprisingly, perform much more consistently when training and testing across multiple users and devices, and even perform as well (and sometimes better) than testing and training on a single user/device. This suggest that dictionaries of accelerometer data can be collected, but there seems to be wide variance for some input types that may affect accuracy.

Movement Noise. Finally, all the results presented previously considered data collected in a controlled movement setting, *i.e.*, while the user was seated at a table. *It is important to know how these models perform if they were predicted from noisy data*, *e.g.*, collected while the user was walking. Although it is likely that an attacker would obtain stable accelerometer data, he/she would also obtain data while the user is in motion. The effects of noisy samples must also be considered if the attacker were to construct a representative corpus.

In Figure 12, we present the results of an experiment that investigates the affect of movement noise. First, we built a model using the data for a single user while they were sitting, and then we tested that model on data collected while the user was walking. Also presented in Figure 12 is the trend line for the performance of the cross-validation while the user is just sitting. Clearly, there is a significant decrease in the inference performance as a result of movement noise. However, what is unclear from this experiment is how a model would perform if it had a large collection of movement noised examples to train on. Unfortunately, we do not have sufficient samples to investigate such a proposition, but it is likely that performance would improve, but would not surpass controlled and movement-stable collection scenarios.

7.2 PIN/Pattern Sequence Inference

The results above model the first attack scenario, where the attacker has a large corpus of labeled data available, and the attacker can apply logistic regression to differentiate input. In this subsection, we consider the second attack scenario, where such a corpus is unavailable, and instead the attacker must infer the larger input by performing a sequence of smaller inferences. For example, we consider an attacker who has a set of labeled data that refers not to the exact PIN/pattern but to examples of single touches of digits or individual swipes. The goal is to link those predictions together using a idden Markov Model (HMM) to infer the whole input.

Single Touch/Gesture Inference. The first step in this process requires showing that the features described previously also differentiate single touch or swipe input. To study this, we segmented the accelerometer data for PINs and patterns based on the recorded touch logs such that features can be extracted based on a single event. As noted previously, the process an attacker may use to segment the data in this manner using just accelerometer data is beyond the scope of this work; however, such segmentation is likely possible, such as described in [34].

We performed experiments for inferring both unigrams and bigrams. A unigram consists of a swipe across a contact point in a pattern, or touching a single digit for a PIN. A bigram consists of a

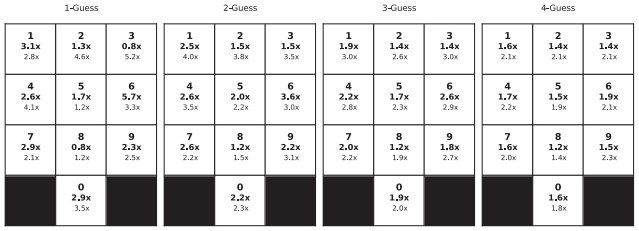


Figure 14: Prediction results for PIN pad as factor greater than random guessing, included (in smaller text) results from *taplogger* [34].

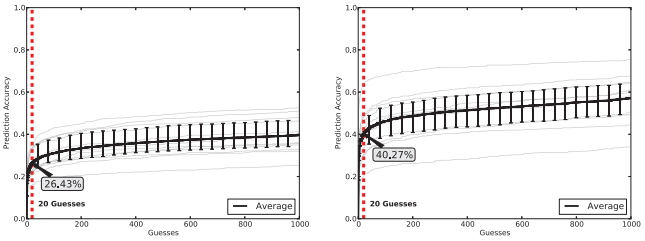


Figure 15: Prediction accuracy for bigram HMM over multiple guesses for patterns (*left*) and PINs (*right*), and the 20 guess threshold is indicated with a dashed line. The shaded trend lines are individual users. Note that PINs outperform pattern prediction, likely due to the limited number of transitions and shorter sequences.

swipe connecting two contact points in a pattern, or two sequential digit presses in a PIN. Thus, there are 9 and 10 possible unigram values for patterns and PINs, respectively, and 72 and 100 possible bigram values for patterns and PINs, respectively. To test the inference capabilities of an attacker, we use the collected accelerometer data and divide it into uni- and bigrams appropriately using the touch information and perform a five-fold cross validation for each user. The average across all users is presented in Figure 13.

Clearly, both uni- and bigram prediction proceeds at a rate well above random chance, with bigrams performing better overall as a factor above random chance. This bodes well for sequence prediction using bigrams. However, when we conducted experiments where we test and train on different users, or when we introduce movement noise, the models fail, either performing a small fraction greater than random chance, or worse. As we will discuss below, when using such models in an HMM, they were unable to infer the input, even after 1000 guess attempts.

Comparison to *Taplogger*. In the case of unigram inference for PINs, we can compare the results of *taplogger* [34] to our own since Xu *et al.* used a numeric number pad, much like PINs. Recall that *taplogger* uses gyroscopic data to infer where on a touchscreen a tap event occurred, while we use accelerometer data. Figure 14 presents the comparisons for four guesses (described as coverage in [34]). Although, *taplogger* performs well, our technique is comparable to *taplogger*'s results, either performing nearly as well, or slightly better, in all instances.

Hidden Markov Model Inference. With models for individual touches or swipes, it is now possible to construct a hidden Markov model (HMM) that selects the most likely (*maximum a posteriori*) set of touch or gesture input. For the experiment, we use a transition matrix trained from a set of 50 PINs and 50 patterns, and use bigram models. We found that prediction results for unigrams were very poor.

The results of the experiment are presented in Figure 15. On the x-axis is the number of guesses (or paths in the HMM attempted) and the y-axis is the prediction result. The most likely path is straightforward to obtain. To generate additional reasonable alternate high scoring paths from the HMM, we order the set of labels at each position by their max-marginal probabilities[4] and employ non-max suppression to get a diverse set of guesses. The details of the technique can be found in [25].

At 20 guesses, the results for both PINs and patterns are very good. Patterns can be inferred with an accuracy of 26%, and PINs with an accuracy of 40%. Note this is a cross-validation for a single user on a single device. We ran similar experiments where we cross train on all users and test on a single user: The results were greatly depressed, and the actual PIN or pattern is rarely predicted. Similarly, we applied these techniques to data from when the users were walking, and, again, we found that the HMM infers input very poorly, predicting with an accuracy far below 1%.

These results suggest that the capabilities of attackers are mixed when limited labeled data is available. In one sense, if the attacker has sufficient training on a single user in a controlled setting, the attacker would likely do very well. However, adverse situations such as movement noise or limited training greatly affects the models, and may even render them completely ineffective.

8. SENSORS AND DEVICE SECURITY

Given these results and previous sensor-based side channel results [6, 7, 22, 24, 34], clearly any effective security mechanisms for touchscreen devices with movement sensors must deny untrusted applications access, at a minimum, to the accelerometer when sensitive touchscreen input is being provided to other applications. At the same time, it may be equally undesirable to restrict access to the accelerometer (and other sensors) when sensitive input operations are *not* being performed. Many legitimate applications are designed to run in the background at all times (*e.g.*, pedometer applications), and preventing such applications from gaining access to the accelerometer at any time, or requiring the user to manually shut them down before performing any sensitive operation, would greatly reduce their appeal.

One approach might be to carefully vet applications that use sensors for malicious behavior before allowing them to be installed or before making them available in application markets. Unfortunately, this approach is logistically impractical at scale. An alternative approach, as exemplified by Google in the Android App Market, is to label applications that access sensors (or other services) using a permission model; however, this is also insufficient because users may either ignore such labels or do not understand their implications.

Another approach may be to restrict the sampling rate of the sensors, as suggested in [24]. However, in our experiments, even with a relatively low sample rate of 20 Hz, prediction accuracy was surprisingly high and on par with devices with sample rates at 50 Hz or more. Such a technique would likely require a reduction in sample rate below the functional level required by legitimate applications.

We propose an alternative strategy: Applications installed by the user that require access to movement sensors, however frivolous they may seem, should be able to use them and use them at the highest sample rate allowed. But, the sensors should be disabled (or untrusted applications denied access to them) whenever a trusted input function – such as password entry – is being performed.

Unfortunately, the security models implemented by current hand-

[4] A max-marginal $m(\ell_i)$ for label ℓ_i at position i in a sequence is obtained by maximizing over the label possibilities in the other positions in the sequence: $m(\ell_i) = \max_{j \neq i} p(\ell_1, \ldots, \ell_k | o_1, \ldots, o_k)$. This can be done for all labels and all positions as efficiently as computing the single most likely assignment [17].

held platforms do not allow temporal access control over sensors; however, context-based security rules proposed in [23] and [9] could be adopted in this way. Currently, applications declare what access they need once (typically when they are first installed by the user or first run), and, from that point onward, have essentially unrestricted, permanent access to everything they asked for at any time they wish.

Although current mobile platforms do not support temporary revocation of sensor access, it could be implemented in a straightforward way, *e.g.*, via a system call available to trusted input functions to obtain and revoke exclusive access to sensors. One approach would be for this system call to cause any untrusted application that requests access to a sensitive sensor to block (or fail) until the sensitive operation has concluded. Alternatively, untrusted applications could simply be suspended for the duration of the sensitive input.

9. CONCLUSION

In this paper we demonstrate that the accelerometer sensor can function as a side channel against secure input, and our results indicate that a surprising amount of information can be inferred, even when movement noise is introduced. We show that there is consistency across users and devices, despite varied sample rates, and the construction of a sensor-reading-to-input dictionary is possible; however, in less controlled settings, such dictionaries may be ineffective. Further, we show that sequence predictions, in the form of a hidden Markov model, can be applied to this problem if insufficient labeled accelerometer readings are available, but such models, again, seem prone to false predictions caused by movement noise and cross-user training.

Given these new results, and previous results using the accelerometer sensor [24] and gyroscopic sensor [6, 34], it is now clear that the security model for on-board sensors on smartphones should be reconsidered. Both the new and previous results should be considered *conservative* estimates of the potential threat: Enhancements to features and larger data sources will inevitably lead to greater fidelity side channels, as was the case for the study of keyboard acoustic side channels from the supervised learning strategies in [1] to the unsupervised learning strategies in [35]. It is clear that applications that have access to the accelerometer sensor should not be able to read from the sensor while the user is providing sensitive input. Current mobile platform permission schemes are not insufficient to specify this; they provide applications with "all or nothing" access to every sensor they might ever need to use. The permission scheme and enforcement mechanism should restrict or allow access to sensors based on *context*: untrusted applications that require access to a sensor should be granted access only when sensitive input operations are not occurring.

References

[1] Dmitri Asonov and Rakesh Agrawal. Keyboard accoustic emanations. In *Proceedings of IEEE Syymposium on Security and Privacy*, 2004.

[2] Adam J. Aviv, Katherine Gibson, Evan Mossop, Matt Blaze, and Jonathan M. Smith. Smudge attacks on smartphone touch screens. In *Proceedings of the 4th USENIX Workshop On Offensive Technologies*, WOOT'10, 2010.

[3] Ling Bao and Stephen Intille. Activity recognition from user-annotated acceleration data. In *Pervasive Computing*, volume 3001 of *Lecture Notes in Computer Science*, pages 1–17. 2004.

[4] Alastair R. Beresford, Andrew Rice, and Nicholas Skehin. Mockdroid: Trading privacy for application functionality on smartphones. In *12th Workshop on Mobile Computing Systems and Applications*, HotMobile'11, 2011.

[5] Joseph Bonneau, Sören Preibush, and Ross Anderson. A birthday present every eleven wallets? the security of customer-chosen banking pins. In *Sixteenth International Conference on Financial Cryptography and Data Security*, FINCRYPTO '12, 2012.

[6] Liang Cai and Hao Chen. Touchlogger: inferring keystrokes on touch screen from smartphone motion. In *Proceedings of the 6th USENIX conference on Hot topics in security*, HotSec'11, 2011.

[7] Liang Cai and Hao Chen. On the practicality of motion based keystroke inference attack. In *Proceedings of the 5th Internaltional Conference on Trust & Trustworthy Computing*, Trust'12, 2012.

[8] Liang Cai, Sridhar Machiraju, and Hao Chen. Defending against sensor-sniffing attacks on mobile phones. In *Proceedings of the 1st ACM workshop on Networking, systems, and applications for mobile handhelds*, MobiHeld '09, 2009.

[9] Mauro Conti, Vu Nguyen, and Bruno Crispo. Crepe: Context-related policy enforcement for android. In Mike Burmester, Gene Tsudik, Spyros Magliveras, and Ivana Ilic, editors, *Information Security*, volume 6531 of *Lecture Notes in Computer Science*, pages 331–345. Springer Berlin / Heidelberg, 2011.

[10] Google Android Development. `http://developer.android.com/reference/android/hardware/SensorEvent.html`.

[11] Splasho Development. Pattern lock pro. `https://market.android.com/details?id=com.splasho.patternlockpro`.

[12] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. Liblinear: A library for large linear classification. *J. Mach. Learn. Res.*, 9:1871–1874, June 2008.

[13] Google Inc. Google wallet. `http://www.google.com/wallet/`.

[14] THQ Inc. Star wars: Lightsaber duel. `http://itunes.apple.com/us/app/star-wars-lightsaber-duel/id362158521?mt=8`.

[15] Rupesh Jain. Pattern encrypt/decrupt upgrad. `https://market.android.com/details?id=PatternEncryptDecryptUpgrade.free`.

[16] M.J. Kearns and U.V. Vazirani. *An introduction to computational learning theory*. The MIT Press, 1994.

[17] D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques*. The MIT Press, 2009.

[18] Jiayang Liu, Lin Zhong, Jehan Wickramasuriya, and Venu Vasudevan. uwave: Accelerometer-based personalized gesture recognition and its applications. *Pervasive Mob. Comput.*, 5:657–675, December 2009.

[19] Jani Mäntyjärvi, Juha Kela, Panu Korpipää, and Sanna Kallio. Enabling fast and effortless customisation in accelerometer based gesture interaction. In *Proceedings of the 3rd international conference on Mobile and ubiquitous multimedia*, MUM '04, 2004.

[20] Philip Marquardt, Arunabh Verma, Henry Carter, and Patrick Traynor. (sp)iphone: decoding vibrations from nearby keyboards using mobile phone accelerometers. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, 2011.

[21] Uwe Maurer, Anthony Rowe, Asim Smailagic, and Daniel P. Siewiorek. ewatch: A wearable sensor and notification platform. In *Proceedings of the International Workshop on Wearable and Implantable Body Sensor Networks*, 2006.

[22] Emiliano Miluzzo, Alexander Varshavsky, Suhrid Balakrishnan, and Romit Roy Choudhury. Tapprints: your finger taps have fingerprints. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, MobiSys '12, 2012.

[23] Machigar Ongtang, Stephen McLaughlin, William Enck, and Patrick McDaniel. Semantically rich application-centric security in android. In *Computer Security Applications Conference, 2009. ACSAC '09. Annual*, ACSAC '09, 2009.

[24] Emmanuel Owusu, Jun Han, Sauvik Das, Adrian Perrig, and Joy Zhang. Accessory: Keystroke inference using accelerometers on smartphones. In *Proceedings of The Thirteenth Workshop on Mobile Computing Systems and Applications*, HotMobile, 2012.

[25] Dennis Park and Deva Ramanan. N-best maximal decoders for part models. In *IEEE International Conference on Computer Vision*, ICCV'11, 2011.

[26] Rio Park. Memorize pattern. `https://market.android.com/details?id=riopark.pattern`.

[27] Kurt Partridge, Saurav Chatterjee, Vibha Sazawal, Gaetano Borriello, and Roy Want. Tilttype: accelerometer-supported text entry for very small devices. In *Proceedings of the 15th annual ACM symposium on User interface software and technology*, UIST '02, 2002.

[28] C. Randell and H. Muller. Context awareness by analysing accelerometer data. In *Wearable Computers, The Fourth International Symposium on*, pages 175 – 176, 2000.

[29] Nishkam Ravi, Nikhil D, Preetham Mysore, and Michael L. Littman. Activity recognition from accelerometer data. In *Proceedings of the Seventeenth Conference on Innovative Applications of Artificial Intelligence(IAAI*, pages 1541–1546. AAAI Press, 2005.

[30] Roman Schlegel, Kehuan Zhang, Xiaoyong Zhou, Mehool Intwala, Apu Kapadia, and XiaoFeng Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *Proceedings of the Network and Distributed System Security Symposium*, NDSS, 2011.

[31] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. Timing analysis of keystrokes and timing attacks on ssh. In *Proceedings of the 10th conference on USENIX Security Symposium*, SSYM'01, 2001.

[32] Bump Technologies. Bump app. `bu.mp`.

[33] Nan Xu, Fan Zhang, Yisha Luo, Weijia Jia, Dong Xuan, and Jin Teng. Stealthy video capturer: a new video-based spyware in 3g smartphones. In *Proceedings of the second ACM conference on Wireless network security*, WiSec '09, 2009.

[34] Zhi Xu, Kun Bai, and Sencun Zhu. Taplogger: Inferring user inputs on smartphone touchscreens using on-board motion sensors. In *Proceedings of the fifth ACM conference on Wireless network security*, 2012.

[35] Li Zhuang, Feng Zhou, and J. D. Tygar. Keyboard acoustic emanations revisited. *ACM Trans. Inf. Syst. Secur.*, 13, November 2009.

Analysis of the Communication between Colluding Applications on Modern Smartphones

Claudio Marforio[†], Hubert Ritzdorf[†],
Aurélien Francillon[‡], Srdjan Capkun[†]

[†]Institute of Information Security
ETH Zurich, Switzerland
{maclaudi,rihubert,capkuns}@inf.ethz.ch

[‡]Networking and Security Group
Eurecom, Sophia-Antipolis, France
aurelien.francillon@eurecom.fr

ABSTRACT

Modern smartphones that implement permission-based security mechanisms suffer from attacks by colluding applications. Users are not made aware of possible implications of application collusion attacks—quite the contrary—on existing platforms, users are implicitly led to believe that by approving the installation of each application independently, they can limit the damage that an application can cause.

We implement and analyze a number of covert and overt communication channels that enable applications to collude and therefore indirectly escalate their permissions. Furthermore, we present and implement a covert channel between an installed application and a web page loaded in the system browser. We measure the throughput of all these channels as well as their bit-error rate and required synchronization for successful data transmission. The measured throughput of covert channels ranges from 3.7 bps to 3.27 kbps on a Nexus One phone and from 0.47 bps to 4.22 kbps on a Samsung Galaxy S phone; such throughputs are sufficient to efficiently exchange users' sensitive information (e.g., GPS coordinates or contacts). We test two popular research tools that track information flow or detect communication channels on mobile platforms, and confirm that even if they detect some channels, they still do not detect all the channels and therefore fail to fully prevent application collusion. Attacks using covert communication channels remain, therefore, a real threat to smartphone security and an open problem for the research community.

1. INTRODUCTION

Modern smartphone operating systems allow users to install third-party applications directly from on-line *application markets*. Given the large number of applications and the number of independent developers, every application cannot be trusted to behave according to its declared purpose. Certain types of malicious behaviors can be detected by inspection and testing while others cannot; malicious applications therefore find their way into application markets [1, 24, 25, 26].

To limit the potential impact of malicious applications, mobile phone operating systems (e.g., Android OS [14], Symbian OS [23], Windows Phone 7 [21]) implement a permission-based security model (also called a *capability model*) that restricts the operations that each application can perform. This model explicitly gives applications the permissions that are required to correctly execute their operations. Recent work by Schlegel et al. [29] introduces smartphone malware that makes use of application collusion over a limited number of communication channels to overcome the security mechanisms put in place by the implemented permission-based model. By establishing communication over a covert or overt channel, applications are allowed to execute operations which the system, based on their declared permissions, should prohibit.

It is important to stress that application collusion attacks on permission-based models are neither a result of a software vulnerability nor related to a particular implementation. Instead, they are a consequence of the basic assumption on which the permission-based model relies: applications can be independently restricted in accessing resources and then safely composed on a single platform. Collusion attacks show that this assumption is incorrect and can be exploited to circumvent the permission-based model. Furthermore, in current systems, users are not made aware of possible implications of application collusion attacks—quite the contrary—users are implicitly led to believe that by approving the installation of individual applications independently, they can limit the damage that an application can cause.

Although the existence of overt and covert channels and thus the feasibility of application collusion on any platform might not be surprising, the implications of collusion are very damaging on mobile platforms: these platforms are designed for personal use, generally store personal information and facilitate the installation of multiple third-party applications. Furthermore, existing security products (e.g., Lookout Privacy Advisor [32]) analyze and report application permissions independently for each individual application. Since they do not consider application collusion, these products do not correctly reflect the collective privacy implications of the applications that the users install.

In this work, we demonstrate the existence of a number of overt and covert channels by implementing them on the

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

Android platform. We then evaluate the severity of the threats posed by application collusion attacks by measuring the throughput and stability of each channel. Our results show that different covert channels, which are generally hard to detect or prevent, can reach throughputs roughly ranging from 3.7 bps up to 3.27 kbps on the Nexus One and from 0.47 bps up to 4.22 kbps on the Samsung Galaxy S, thus posing a serious threat to privacy on modern smartphones.

While it is shown that overt channels on mobile smartphones may be detected or restricted using taint analysis [10], policy enforcement [3, 4], better sandboxing and by reducing access to some APIs, we show that these approaches fail to detect most covert channels. This is consistent with research carried out in the 1970's where it has been shown that covert channels in computer systems are hard to prevent [9, 20]. To evaluate the effectiveness of contemporary tools, we tested both TaintDroid [10] and XManDroid [4] confirming that they do not detect all channels and therefore fail to fully prevent application collusion. This shows that application collusion attacks remain a real threat on modern smartphone platforms. Finally, we propose ways of preventing or limiting some of these channels.

In summary, the contributions of this paper are the following. (1) We demonstrate the practicality of application collusion attacks by implementing several communication channels on the Android platform. (2) We measure and report the throughput of implemented communication channels highlighting the extent of the threat posed by such attacks. (3) We confirm that two recently proposed architecture modifications and tools that deal with overt and covert channels discovery, TaintDroid [10] and XManDroid [4], still fail to detect several of the implemented channels. (4) We propose countermeasures that, if not eliminate, limit the throughput of selected communication channels between the applications.

The rest of the paper is organized as follows. In Section 2 we present the problem statement with a classification of communication channels. We then show the results of our study of communication channels in the Android OS in Section 3. The analysis of TaintDroid and XManDroid in the setting of our testing framework is detailed in Section 4. We then discuss current mitigation techniques and their limitations in Section 5. Finally, we present related work in Section 6 and conclude in Section 7.

2. PROBLEM STATEMENT

The goal of this work is to understand the threat posed by colluding applications on modern smartphones. In particular we investigate the feasibility and the practicality of multiple communication channels in terms of throughput, bit-error rate and required synchronization. Figure 1(a) illustrates an example channel between two applications. On the left, the *ContactsManager* application has access to private data on the device but not to the network (later in the work referred to as the *source* application), on the right, the *Weather* application having access to the network but no direct access to the private data (later in the work referred to as the *sink* application). The two applications can create a stealthy communication channel to share data. Figure 1(b) illustrates an interesting covert channel that can be created between an application and the Browser which does not require any extra application to be installed on the device. We will describe this channel in more detail in Section 3.3.

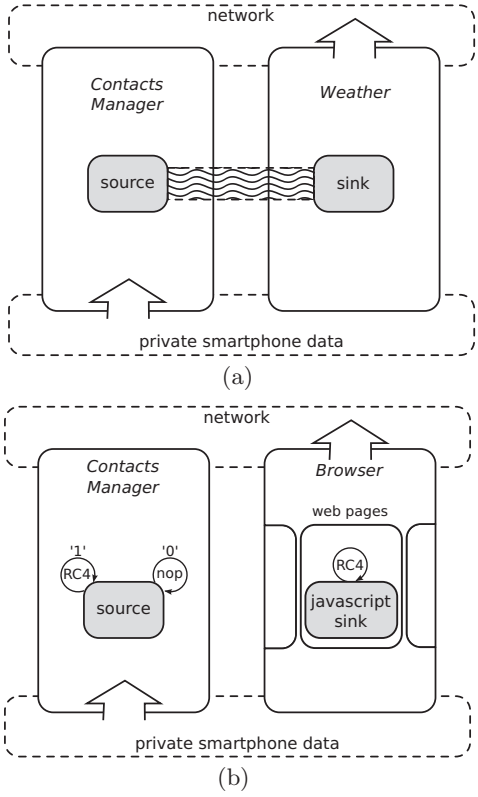


Figure 1: Figure (a) shows a generic example of collusion between the *ContactsManager* and the *Weather* applications through a stealthy (overt) communication channel. Figure (b) shows a (covert) timing channel between an application and the browser working through the use of dummy RC4 operations.

2.1 Channels Classification

We classify communication channels based on their implementation on current smartphone architectures as follows:

- **Application.** This is the level of the API that an operating system provides to the developers (e.g., Android's Java API, Windows Phone 7 C # / Silverlight APIs, iOS's Objective-C API). Access and usage of these communication channels may be easily controlled. We consider these channels as *high*-level.
- **OS.** This is the level of the operating system that is exposed through native calls that exploit information present in the operating system. We believe that at this level some channels are impossible to close, others, if closed, could hamper backward compatibility severely.
- **Hardware.** This is the level that is exposed through exploiting hardware functionalities of the smartphone. It is highly dependent on individual hardware specifications of smartphone models. These communication channels cannot be closed without severe performance degradation of the system. We consider these channels as *low*-level.

Different levels usually also imply different throughput and stealth. In particular, we notice that throughput is usu-

Overt Channel	Throughput (kbps)	
	Nexus One	Samsung Galaxy S
UNIX Socket Communication	340.45 (± 154.02)	34.78 (± 11.39)
Internal Storage	292.03 (± 50.06)	32.60 (± 8.47)
Shared Preferences	75.81 (± 6.83)	31.00 (± 2.75)
Broadcast Intents	40.58 (± 8.41)	26.74 (± 4.88)
External Storage †	11.55 (± 1.10)	6.12 (± 3.95)
System Log ‡	2.94 (± 0.03)	2.14 (± 0.11)

† Requires extra `WRITE_EXTERNAL_STORAGE` permission.
‡ Requires extra `READ_LOGS` permission.

Table 1: List of our implemented *overt* channels in the Android OS with corresponding throughputs (with the 95% confidence intervals in parenthesis). The displayed values are averaged over 10 runs for both the Nexus One and the Samsung Galaxy S. The "System Log" is a new channel that we engineered and for which we did not find references in the open literature.

ally directly proportional to the level, with higher throughput associated to *high*-level communication channels. Stealth (i.e., the difficulty to detect a communication channel), on the other hand, is usually inversely proportional to the level, with stealthier channels associated to *low*-level communication channels.

3. OVERT AND COVERT CHANNELS IN ANDROID

We explore possible covert and overt channels on Android smartphones. We analyzed some known channels and identified a number of new channels specific to smartphones not yet presented in the literature.

To analyze overt and covert channels, we implemented a framework to measure the throughput, the bit error rates and the synchronization times for each implemented communication channel. The results of our study are presented in Tables 1 and 2.

The values shown in both tables are averaged over 10 independent runs for each implemented channel executing on a Nexus One or a Samsung Galaxy S smartphone. During the tests the phone was running on battery power and not charging. Each time the *source* application tries to send 4, 8 and 135 byte (to mimic the transfer of contact information as explained in Section 3.4) messages to the *sink* that, if the channel is open, would record them successfully. For each covert channel that requires tight synchronization between the *source* and the *sink* application (i.e., timing channels), we implement a synchronization protocol and run it before starting to send data. In general the synchronization protocol is used to estimate the noise present in the system when the applications want to share data and to correctly start the measurements to exchange data at the same time. Such a protocol is implemented on top of the covert channel (i.e., using the same mechanism) and allows us to reach higher accuracy. A covert channel's accuracy is measured in bit error rate, with perfectly accurate channels reaching 0% bit error rate during transmission. While synchronization time values are reported in Table 2 for completeness, we believe that they can be further optimized to yield overall slightly faster communication channels.

3.1 Overt Channels

We now briefly describe the implementation of overt channels to give an intuition of how they work.

Shared Preferences (Application): the *sink* application uses an API to create an Android preference XML file that is world-readable and world-writable. The *source* application writes ASCII data to it and the *sink* reads it. This channel does not require any synchronization to operate as the two applications do not need to be run simultaneously.

Internal Storage (Application): the *source* application writes a world-readable file to the internal storage, the *sink* application reads its contents. Similarly the *External Storage* simply uses a file on the external storage. For the external channel to work, the *source* application requires an extra permission: `WRITE_EXTERNAL_STORAGE`. Again, similar to the *Shared Preferences* communication channel, these channels do not require synchronization between the applications.

Broadcast Intents (Application): the *source* application communicates by adding private data as extra payload to a broadcast message sent to the system. Broadcast intents are a particular type of messages that are used in the Android OS to enable one form of communication between applications. The operating system, upon receiving such a message with its payload, broadcasts it to all the applications that requested to be notified when such a message is received (i.e., by registering themselves for a particular ID that is used to identify the message). The *sink* application registers itself with the system and receives the message sent by the *source*. While both applications need to be running at the same time, no synchronization is required in order for the channel to work.

System Log (Application): the *source* writes a specially-crafted message to the system log that the *sink* then reads to extract the information. The extra `READ_LOGS` permission is required by the *sink* application in order to be able to read the system logs. Messages longer than 4000 characters must be split and binary data must be encoded, because data is otherwise lost when inserted into the log. Given that the log has a finite number of entries that are held at any time, the *sink* application must be activated before the message sent by the *source* is deleted. Alternatively, the *source* could repeatedly insert the message at time intervals to increase the chance that the *sink* receives it. Potentially the channel can be rendered stealthy by filling the log with seemingly meaningful logging data after the communication takes place.

UNIX Socket Communication (OS): the *source* sends the data through a UNIX socket that the *sink* application opened. For this channel to work correctly, both applications must be simultaneously active.

3.2 Covert Channels

We now describe the covert channels that we implemented and measured. As the storage of these channels is not persistent, all these channels are synchronous. This means that before starting to exchange data over the channel a synchronization protocol between the *source* and the *sink* must be run in order to achieve better accuracy during the data-exchange phase. For channels where accuracy is not specifically stated, our implementation reached perfect accuracy.

Single and Multiple Settings (Application): the *source* modifies a general setting on the phone and the *sink* reads it as described in [29]. Multiple settings can be changed at the same time to achieve higher throughput. Most settings in Android can be changed and read without requesting any permissions. This particular covert channel can be closed by disabling or requiring extra permissions in order to change particular settings.

Type of Intents (Application): the *source* sends a broadcast message (similar to the *Broadcast Intents* overt channel) to the *sink* and encodes the data to be transmitted into the type of the intent (i.e., flags, action, particular extra data), rather than directly exchanging the data as the extra payload of the message. In contrast to the similar overt channel that uses *Broadcast Intents*, this covert channel is not detectable by tainting mechanisms or similar solutions. The *sink* application still needs to register with the system in order to receive the intents.

Automatic Intents (Application/OS): the *source* modifies particular settings (i.e., the vibration setting [29]) that trigger automatic broadcasts by the system to all applications that registered to be notified when such a change happens. The *sink* receives the messages and infers the data depending on the contents of the received broadcasts. For instance, changing the vibration setting of the phone triggers a broadcast which contains 1-bit of information (vibration on equals to 1, vibration off equals to 0).

Threads Enumeration (OS): the *source* spawns a number of threads and the *sink* reads how many threads are currently active for the source application by looking into the `/proc` directory of *source*. This particular covert channel can be closed by controlling application access to the `/proc` filesystem or by mediating the access through a system service.

UNIX Socket Discovery (OS): the *source* uses two sockets, a synchronization socket and a communication socket. The *sink* checks if the *source* communication socket's state is open, and infers the transferred bit. The synchronization socket is open if the communication socket can be checked.

Free Space on Filesystem (OS): the *source* application writes or deletes data on the disk to encode the information for the *sink*. The channel throughput depends on the noise in the system; for example the sink application could infer a larger amount of information depending on how many free blocks are available. The data presented in Table 2 was generated by having the *source* allocate three blocks to encode a '1' and clearing three blocks to encode a '0'. The *sink* checked the available blocks in the system at predefined time intervals (75ms for the Nexus One and 100ms for the Galaxy S). The current implementation yields bit-errors percentages between 0.01% (Nexus One) and 0.03% (Samsung Galaxy S). A possible solution for preventing this channel is to enforce a quota on the available space for each application (that could potentially vary depending on the number of applications on the system) and report the free blocks remaining of each quota rather than the free blocks in the overall system.

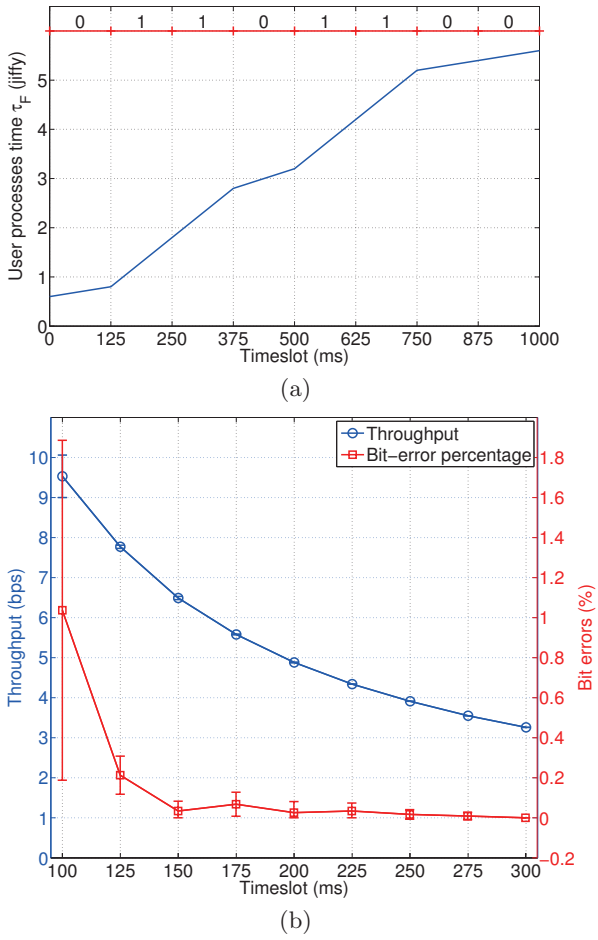


Figure 2: Figure (a) shows an exemplification schematic rise of the value τ_F (number of jiffies spent for every user process) over time when sending the bits written on the top. Figure (b), the graph shows the trade-off between throughput and accuracy (measured in bit errors) for the `/proc/stat` channel. Values are averaged over 5 independent runs.

Reading `/proc/stat` (OS): the *source* application performs some computations, while the *sink* monitors the processor usage statistics. These are available in the `/proc/stat` virtual file where the Linux kernel provides information about the current system load (as the number of jiffies used for all user processes). A schematic representation of how the values (τ_F) read in `/proc/stat` change depending on the bit that the *source* wants to send is presented in Figure 2(a). The overall idea is that sending a '1' causes, in the values read, a steeper slope than sending a '0'. Figure 2(b) presents the trade-off between throughput and accuracy of this channel. Other channels behave similarly to this one, with higher throughput resulting into lower accuracy. The current implementation yields bit-error percentages between 0% (Samsung Galaxy S) and 0.10% (Nexus One). Similarly to the *Threads Enumeration* covert channel, this channel could be closed by preventing read access to the `/proc` filesystem.

Covert Channel	Throughput (bps)				Synchronization (ms)			
	Nexus One		Samsung Galaxy S		Nexus One		Samsung Galaxy S	
Type of Intents	3350.85	(± 134.11)	4324.13	(± 555.32)	716.8	(± 168.2)	473.0	(± 249.0)
UNIX Socket Discovery	2610.92	(± 305.25)	1647.78	(± 170.70)	5.2	(± 0.8)	13.9	(± 2.2)
Multiple Settings	239.76	(± 9.41)	284.91	(± 1.90)	314.9	(± 21.8)	302.1	(± 11.0)
Threads Enumeration	157.73	(± 0.97)	139.39	(± 7.40)	71.6	(± 7.1)	110.1	(± 8.8)
Automatic Intents [29]	51.38	(± 0.41)	90.67	(± 0.39)	1083.2	(± 75.1)	435.1	(± 180.8)
Single Settings [29]	46.88	(± 0.31)	65.89	(± 0.73)	267.5	(± 3.2)	273.4	(± 11.9)
Free Space on Filesystem	13.07	(± 0.00)	9.80	(± 0.00)	1038.2	(± 5.1)	1442.7	(± 15.6)
Reading `/proc/stat`	7.82	(± 0.00)	3.26	(± 0.00)	6923.4	(± 8.1)	16669.2	(± 48.7)
Processor Frequency	4.88	(± 0.00)	0.47	(± 0.09)	8203.9	(± 7.2)	78866.1	(± 9156.8)
Timing Channel	3.70	(± 0.00)	3.69	(± 0.01)	10286.8	(± 16.1)	68057.6	(± 105259.4)

Table 2: List of implemented *covert* channels in the Android OS with their corresponding throughput (95% confidence intervals are shown in parenthesis). The displayed values are averaged over 10 runs for both the Nexus One and the Samsung Galaxy S. Channels listed in bold are new channels that we engineered and for which we did not find references in the open literature. In the synchronization column we present the time required to run the synchronization protocol before starting to transmit data through the channel.

Timing Channel (Hardware): the data transmission between the source and the sink is performed by varying the load on the system. The *source* runs CPU-intensive tasks to send the bit '1', on the other hand it does not perform any CPU-intensive operation to send the bit '0'. The *sink* continuously runs computation-intensive operations and records the time required to complete them. The *sink* uses this time to infer the presence of computation by the *source* thus inferring the transmitted bit. For reliable differentiation of bits based on the time, an initial *learning period* is used to benchmark the system behavior. Finally, to eliminate the noise in the system, we use a majority vote (out of five measurements) at the *sink* to decide the value of a particular bit depending on a threshold value updated with a moving average. In our implementation, the time difference between transmitting a '1' and a '0' is approximately 6 ms in the case of the Nexus One. The current implementation yields bit-errors percentages between 0.10% (Nexus One) and 0.05% (Samsung Galaxy S).

Processor Frequency (Hardware): this channel is an improvement over the basic *Timing Channel*; in this particular instance we take into account Dynamic Frequency Scaling (used on the smartphones that we tested) to improve the throughput and reduce the synchronization time (for the Nexus One). While the *source* behavior remains the same as in the case of the *Timing Channel*, the *sink* instead monitors the trend of the processor frequency by repeatedly querying it from the system and thereby decodes the current bit. Afterwards the *source* waits a fixed amount of time to allow the CPU to "slow down" again before the next bit transmission is started. The current implementation yields bit error percentages between 0.14% (Nexus One) and 4.67% (Samsung Galaxy S).

3.3 Communication Channel With External Agents

We extend the concept of colluding applications and consider the scenario in which there is only one application installed on the system that has access to private data and wants to disclose it to a third-party web service without requesting the permissions to connect to the network. Furthermore, we want to ensure the successful transmission of the private data through a channel that is hard to detect. Here, the colluding *sink* application resides on a web page executing intensive JavaScript operations, that is opened within the system browser. The phone will show the page on the screen, therefore, to decrease detection by the user, the operation can be carried out when the phone screen is off (for example, during night time). To reach the *sink*, the *source* application uses a covert timing channel similar to the *Processor Frequency* covert channel. However the *sink* cannot directly query the processor frequency, as it is inside the JavaScript sandbox. Such channel is visualized in Figure 1(b).

We have implemented and tested this proposed covert channel as follows: depending on the current bit to transfer, the *source* either tries to increase the processor frequency or sleeps. Afterwards the *sink* measures how many dummy RC4 operations it can perform in a fixed time period, thereby getting the processor frequency and the transmitted bit. The possibility to use the browser to send private data has been described in [19] but their proposed method is easily detected by flow-tracking techniques (such as TaintDroid). Our proposed covert channel—while having a low throughput of roughly 1.29bps on the Nexus One—is also much harder to detect and cannot be detected by today's tools.

3.4 Results of the Analysis

The experiment results reported in Tables 1 and 2 indicate that the attacker's choice of one channel over another, depends on the nature and size of data that needs to be transmitted between applications. For example GPS coordinates usually consist of two floating point numbers (represented by 32 bits of data), in contrast contacts, for example, might have a varying number of characters: in order to simulate a few full names and corresponding phone numbers we transmit 135 bytes of information.

Given a rough estimate of the size of the data that can be shared between applications, we conclude that even the covert channels with low throughput, such as the *Timing* or the *Processor Frequency* channels (respectively at around 3.70 and 4.88 bps) enable the sharing of reasonable amounts of data on the smartphone. For example, exchanging GPS coordinates requires roughly 19.4 or 14.8 seconds respectively; sharing 135 byte contacts requires roughly 304.9 or

231.1 seconds respectively. Covert channels with higher throughput, such as the *Type of Intents* or *UNIX Socket Discovery*, reaching up to 4324.13 and 2610.92 bps, enable the exchange of GPS coordinates or contact information in less than a second.

Another interesting result of the analysis is that most channels, when tested on the more powerful Samsung Galaxy S, did not perform better than on the Nexus One. For CPU-bound channels these results come from the fact that Samsung ships the device with a larger number of active services which influence the different channels. For channels based on Processor Frequency it is based on the fact that there is a different frequency governor. For IO-bound channels these results come from the fact that the Samsung device uses a Samsung-developed file system rather then the standard YAFFS2 used on the Nexus One.

Overall, the results show that application collusion attacks, through the usage of different communication channels depending on the amount of data that needs to be transmitted, are a realistic attack and therefore a serious threat.

4. ANALYSIS OF EXISTING TOOLS

In this Section we test two recently proposed tools that try to solve the information leakage on modern smartphones, in particular TaintDroid [10] and XManDroid [4]. We use the Nexus One as the test phone where we successfully install both tools and report the findings.

4.1 TaintDroid

TaintDroid [10] tries to track information flows within an application and between applications; it is implemented as a modification of the Android operating system. Using dynamic taint-tracking, the modified OS follows the information flow of tagged data, that is, data which is generated from sources of private information, including the user contacts and the GPS location.

Inside the Dalvik VM, TaintDroid employs variable tracking and propagates taint through primitive data types, exception handling routines and array lookups. Tainting information, though, does not follow through in native code (such as JNI native libraries) execution. Due to this limitation, at the moment of writing, trying to use native libraries not residing in the `/system` folder results in an application crash. Additionally, taint is propagated through IPC messages, by performing message-level tracking.

Whenever tainted data reaches a sink (such as the network), a notification is shown informing the user about the application that is leaking data, the originating data class and the network transmission. Interestingly enough, the implementation of TaintDroid notifies the user when the *sink* application uses Java's `HttpURLConnection` to send the data off the device, but no detection happens when it uses a UDP connection (i.e., through the Java `DatagramSocket` class). We believe this is just an implementation detail overlooked by the authors rather than a design flaw of the proposed solution.

In our study, TaintDroid was able to correctly report the transmission of sensitive data for the following overt channels: *Internal Storage* and *Broadcast Intents*. The *External Storage* channel was not detected: this happens because taint information is propagated using extended attributes and external storage uses the FAT file-system which does not support them. Surprisingly the remaining overt channels (*Shared Preferences* and *System Log*), which should be detected by TaintDroid were not detected. Further analysis shows that the implementation of the logging mechanism in Android OS is carried out in native code (i.e., C). As previously stated, TaintDroid is not currently capable to extend tagging to native code and therefore cannot detect this channel.

Given that the authors explicitly state that the TaintDroid mechanisms can be circumvented through the use of implicit flows, it is clear that the covert channels implemented in our framework remain undetected due to their bit-wise nature. To remove the taint from tainted variables, such that higher-throughput overt channels can be used successfully, we propose four different techniques. We implement each technique and test it on a Nexus One to report the throughput.

n-way Switch Statement [6]: an n-way Switch Statement can be used to strip the taint off $log(n)$ bits. The Switch Statement reads the tainted value and writes the corresponding constant into a new untainted memory location. The taint does not propagate, because constant values are used. We measured the throughput of this technique to be, roughly, 27.65 Mbps (megabits per second).

Java Exception Handling: Here we encode a tainted bit in the existence of a Java exception. If the tainted bit is '1', an exception is thrown that causes the untainted bit as well to be set to '1' by the exception handler [22]. We measured the throughput of this technique to be, roughly, 107.42 kbps (kilobits per second).

File-based: This technique encodes the tainted bit in the existence of a file inside the application's private directory. Depending on the tainted bit a special file is either created or not. The untainted bit is set depending on the results of the following existence check for the special file. We measured the throughput of this technique to be, roughly, 680 bps (bits per second).

Timing-based: The application's own execution time encodes the tainted bit in this example. The application delays its own execution by sleeping in order to signal a one. Timing measurements determine the value of the untainted bit. We measured the throughput of this technique to be, approximately, 98 bps (bits per second).

Given the throughput of each untainting technique, and the fact that covert channels remain undetected by TaintDroid, we conclude that employing a dynamic flow-tracking technique does not prevent application collusion attacks.

4.2 XManDroid

XManDroid was first presented in [3] and later extended in [4]. It aims at implementing different techniques to successfully mitigate the problems of *confused deputy* attacks and direct *application collusion* attacks.

The authors propose a security framework to enable policy enforcement at different system levels on Android. The instantiation of the framework extends various parts of the Android OS, in particular they port and extend TOMOYO Linux [16]. The security framework modifies Android *reference monitor* to check for direct IPC calls at runtime between applications and indirect communication through Android system components (i.e., the settings manager). Furthermore, kernel-level MAC (Mandatory Access Control) monitors access to different resources such as the file system, UNIX sockets and internet sockets. System policies are ex-

pressed in a high-level language and specify which flows are to be denied.

The prototype with which we experimented was able to block a subset of channels that are potentially detected by XManDroid. In particular the prototype successfully detected all the *overt* channels except the *System Log* channel. It also successfully detected the *Type of Intents* and *UNIX Socket Discovery* covert channels, as they work over explicit communication between applications. Further, the *Reading* `/proc/stat` and *Threads Enumeration* covert channels are detected by the fact that they work by accessing the `/proc` file system, blocked by the TOMOYO Linux access control.

Similarly, it is safe to assume that XManDroid would be able to detect the *Broadcast Intents* and *UNIX Sockets Communication* channel, because they work over explicit communication between applications. The *System Log* channel could also be detected because it works over simultaneous access of a shared file (i.e., similarly to the *storage*-based channels that are detected).

This leaves our analysis with a small subset of covert channels that are not detected by XManDroid: *Free Space on Filesystem*, *Processor Frequency* and finally *Timing Channel*. In particular the last two that are *hardware-level* communication channels are not in the scope of XManDroid.

One limitation of using XManDroid and similar tools is that they might report false-positive results when two non-malicious applications try to share legitimate data, as the communication is blocked even if non-sensitive data is shared (i.e., XManDroid is agnostic of the transmitted data). While the authors claim that this is not an issue because non-malicious applications generally do not tend to share data, it might be interesting to understand if it is possible to render XManDroid data-gnostic. Similarly, restricting access through policies to parts of the system (i.e., `/proc`) might result in some applications to malfunction in case they rely extensively on access to such resources. Finally, XManDroid works by specifically adding hooks to system functions or to the kernel as new channels are discovered, and is therefore a reactive solution.

Given that some channels remain undetected under existing state-of-the-art tools, we conclude that application collusion attacks remain a threat and stress that research should focus on closing these (obviously harder to deal with) channels.

5. MITIGATION TECHNIQUES AND THEIR LIMITATIONS

Solving the confinement problem, and in particular closing all possible covert channels in a system, is known to be a difficult problem [9, 20]. It is further complex in the case of smartphones, where performance, application markets openness and exposed API features are key to user and developer adoption. Mitigation can be achieved either at design time (by reducing access to sensitive APIs or by limiting communication possibilities) or by analyzing static and dynamic properties of applications and their interactions off-line or at run-time.

5.1 Design Time Mitigation Techniques

General Purpose Techniques. There are a number of techniques that could be considered by smartphone operating system designers:

User control on private data access: As in Windows Phone 7, involving user action on each data access helps to mitigate the impact of colluding applications (and more generally, malicious applications). However, this also limits applications capabilities; for example, in such an environment, it is impossible for third-party developers to create applications that perform automated backups of private data.

Limiting APIs: When designing APIs exposed to third-party developers, designers should carefully consider the possibility that the API may create a communication channel between applications. If an overt or covert channel is found, it should be either mitigated or its access should be controllable through the system's policies.

Limiting Multitasking: Reduces the possibility of covert channels resulting from competition for access to resources (CPU time, cache and bus contention). However, this limits the diversity of applications that can be implemented on the system.

Application Review: Performed to detect colluding applications before publication of applications on the markets. However, this approach requires dedicated techniques to detect application collusion.

Policy-Based Installation Strategy: Could be used in a corporate scenario where installation of applications can be limited through some policies e.g., deny access to applications that read contacts.

Application-Level Channels. Communication channels constructed at this level are dependent on the APIs exposed by the underlying operating system. Careful design of permissions used to access data sources as well as data sharing points (i.e., sharing of files or preferences, settings, broadcast intents) could draw attention towards applications that require an excessive number of permissions. Furthermore, some of these channels could be closed by removing unnecessary APIs after an analysis of the used and unused ones. This would enable tighter security while maintaining a reasonable amount of freedom for the developers. For instance, the *System Log* channel can be closed by allowing access to the log file only when "USB Debugging" is active. This is a mode to which the phone switches to when connected to a PC, for instance, for development purposes.

Operating-System-Level Channels. Covert channels that can be established at this level might not be detectable by information flow analysis and their prevention requires further investigation. Such channels usually utilize mechanisms offered by the underlying kernel (i.e., sockets, threads, child processes) and therefore, removing such functionality by preventing developers from using it might impede certain applications and their potential optimization. Other system information made available (for example, through the `/proc` filesystem by the Linux kernel) could be restricted (for example, as done in GRSecurity [15], TOMOYO Linux [16] or SEAndroid [31]) or mediated by operating system services that could directly control access to such information.

Hardware-Level Channels. Covert channels (e.g., timing) established at this level are the usually hardest to remove without serious performance degradation or functionality impact. Solutions, such as preventing multitasking or flushing caches between process scheduling, limit the overall performance or responsiveness of the system and increase its power consumption. Furthermore, common data tainting or information flow control techniques are ineffective in this scenario since communication happens at the bit-level of

the transmitted data. Closing these channels requires novel approaches, e.g., design of information flow secure systems from the bottom up [33], however redesigning current smartphone systems from scratch is likely to have a prohibitive cost.

One possible solution for timing-based channels (similarly proposed for different systems in [17]) is to add a new permission to the Android OS (for example, it can be named `REQUIRE_PRECISE_TIMING`). Applications requiring such permission, upon requesting timing information, would be given precise timing information (i.e., games require precise timing for physics engines or graphics display). Applications without such permission would be presented with a rough estimate of timing (i.e., ± 5 seconds). This modification would disrupt the correct functioning of communication channels that require precise timing for their operation such as, in our reported channels, the *Timing Channel* and the *Processor Frequency* channel. For example, in our implementation the *Timing Channel* works over 6 ms differences (to send a '1' or a '0'), therefore if the system would report time to applications at a granularity higher than 6 ms the channel would be disrupted.

If further analysis shows that precise timing is indeed required for the correct functioning of a large number of applications, another viable solution to disrupt timing channels is to limit the number of times applications can request timing information.

5.2 Application Analysis Techniques

Black-box Analysis. One strategy in trying to detect collusion is to add a data monitor between separate applications on the device. This would remove the need to detect covert channels by only monitoring data leakage itself. In such an architecture, when data from one application is used, the monitor would store it (or a fingerprint of it) and track the data sent to the colluding application. While this approach seems promising, it is inherently limited: malware can encode data in a way that it leaves the mobile device still encoded (e.g., encryption using a public key), defeating the monitoring. While it is clear that black-box analysis may detect some trivial attempts to evade the system security policy, it clearly does not provide a complete solution.

Exclusive access to sensitive resources. Techniques to limit access to sensitive resources (e.g., the microphone) from third-party applications when a sensitive operation is ongoing (e.g., a phone call), as presented in [29], only prevents malware from accessing that particular data at that instant. Such techniques cannot be applied generally: for example, access to the GPS data would always be considered a privacy invasive operation and therefore would never be allowed.

Offline application analysis. Since colluding applications are communicating on an unexpected channel, it is likely that when colluding applications are executed simultaneously on a device, they would show a different behavior than when executed independently. For example, they would detect each others presence and engage into communication over a covert channel. Behavioral analysis could be used to detect such a change of behavior, for example executing applications on an emulator alone or in pair and comparing execution traces and coverage. However, given the vast number of potential pairs of colluding applications, this solution does not scale. This can be addressed by strategies which include evaluating applications according to their popularity or according to "replicated" installations [28].

6. RELATED WORK

The Confinement Problem and Covert Channels. Lampson first described the confinement problem [18] as the problem of preventing unauthorized communication, over overt or covert channels, between two subjects on a system. It is recognized to be a difficult problem in practice, Denning and Denning state that "Cost-effective methods of closing all covert channels completely probably do not exist" [9].

While overt channels can be managed by security policies, *covert channels* are communication channels built from resources that are not intended for communication, and so they cannot be mitigated with the same techniques. Covert channels were also used to perform covert communications over networks [27, 12], however in this work we mainly focused on inter-process covert channels. Inter-process covert channels can be classified as either software (sometimes referred to as TCB channels) or hardware (also known as fundamental channels) and communicate over timing or storage channels. However, this distinction is more empirical than theoretical [13].

Software covert channels can be mitigated by a careful analysis of the usage of visible and alterable variables used by system calls [34] or using a formal model for analyzing programs [30] using a semi-automated technique. However, hardware-related covert channels (e.g., timing, competition to access resources, paging) are difficult to prevent and recent processor designs have been shown to increase the number and efficiency of covert channels [36].

As an example, multi-core application processors are already available for smartphone devices, which would render covert channels over cache highly reliable [36]. Possible mitigation techniques include using fuzzy time [17] and preventing multitasking.

Permission-Based Security. A significant amount of work has been performed, in the past few years, on the Android platform and specifically on the permission-based model [2, 5, 7, 8, 24, 26, 35]. Barrera et al. present an empirical methodology for the analysis and visualization of the permission-based model, which can help in refining the permission system [2]. The *Kirin* tool [11] uses predefined security rule templates to match dangerous combinations of permissions requested by applications. As *Kirin* analyzes individual applications, colluding applications would not be detected by such policies. Saint [26] allows run-time control over communication among applications according to their permissions. In [5], Burns discusses possible unchecked information flows due to applications that use *Broadcast Intents* without proper permissions checking.

Soundcomber[29] introduces a proof of concept malware based on application collusion for Android smartphones. The authors foresee using the microphone of the device to harvest sensitive information, such as credit card numbers, by detecting voice and tone patterns. The information is then sent over an application with the necessary permissions to send it through the internet through means of covert communication channels. The channels presented by the authors use globally-available settings (vibration, volume, screen lock, etc.) or file locks. In contrast, we presented a wide range of channels and carried out an analysis of their behavior in terms of throughput and accuracy through our framework.

7. CONCLUSION

We demonstrated that application collusion attacks against the permission-based mechanisms used on modern operating systems for mobile devices, such as Android OS, are a serious threat given different implementation of communication channels at different system levels. The results of the throughput measurements for each channel show that even covert channels with low throughput are still sufficient to exchange possibly private information stored on a smartphone. Finally, we confirmed that proposed and implemented techniques and tools do not provide a complete solution against different communication channels and are therefore insufficient to prevent application collusion attacks, which remain an open problem for the research community.

Acknowledgments

This work was partially supported by the Zurich Information Security Center (ZISC). It represents the views of the authors. We would like to thank Sven Bugiel and the team behind XManDroid for giving us the possibility to test XManDroid and for productive discussion.

8. REFERENCES

[1] J. Anderson, J. Bonneau, and F. Stajano. Inglorious Installers: Security in the Application Marketplace. In *Workshop on the Economics of Information Security*, WEIS '10, 2010.

[2] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji. A methodology for empirical analysis of permission-based security models and its application to Android. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 73–84, New York, NY, USA, 2010. ACM.

[3] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi. XManDroid: A new Android evolution to mitigate privilege escalation attacks. Technical Report TR-2011-04, Technische Universität Darmstadt, April 2011.

[4] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on Android. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium*, NDSS '12, February 2012.

[5] J. Burns. Developing secure mobile applications for Android. `https://www.isecpartners.com/files/iSEC_Securing_Android_Apps.pdf` (accessed October 2012), 2008.

[6] L. Cavallaro, P. Saxena, and R. Sekar. On the limits of information flow techniques for malware analysis and containment. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '08, pages 143–163, Berlin, Heidelberg, 2008. Springer-Verlag.

[7] A. Chaudhuri. Language-based security on Android. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, pages 1–7, New York, NY, USA, 2009. ACM.

[8] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on Android. In *Proceedings of the 13th International Conference on Information Security*, ISC'10, pages 346–360, Berlin, Heidelberg, 2011. Springer-Verlag.

[9] D. E. Denning and P. J. Denning. Data security. *ACM Comput. Surv.*, 11(3):227–249, Sept. 1979.

[10] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[11] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 235–245, New York, NY, USA, 2009. ACM.

[12] C. G. Girling. Covert Channels in LAN's. *IEEE Transactions on Software Engineering*, 13(2):292–296, 1987.

[13] V. Gligor. A guide to understanding covert channel analysis of trusted systems, version 1 (light pink book). NCSC-TG-030, Library No. S-240,572, November 1993. National Computer Security Center, TCSEC Rainbow Series Library.

[14] Google. Android OS (up to version 2.3.7). `http://developer.android.com/`.

[15] GRSecurity. The GRSecurity project. `http://grsecurity.net/features.php`.

[16] T. Harada, T. Horie, and K. Tanaka. Task oriented management obviates your onus on linux (TOMOYO Linux). Linux Conference, 2004.

[17] W.-M. Hu. Reducing timing channels with fuzzy time. In *Proceedings of IEEE Computer Society Symposium on Research in Security and Privacy*, pages 8–20, May 1991.

[18] B. W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, Oct. 1973.

[19] A. M. Lineberry. These aren't the permissions you're looking for. BlackHat USA, August 2010.

[20] S. B. Lipner. A comment on the confinement problem. In *Proceedings of the 5th ACM Symposium on Operating Systems Principles*, SOSP '75, pages 192–196, New York, NY, USA, 1975. ACM.

[21] Microsoft. Security for Windows Phone 7. `http://msdn.microsoft.com/en-us/library/ff402533%28v=VS.92%29.aspx` (accessed October 2012).

[22] S. K. Nair, P. N. D. Simpson, B. Crispo, and A. S. Tanenbaum. A virtual machine based information flow control system for policy enforcement. *Electron. Notes Theor. Comput. Sci.*, 197(1):3–16, Feb. 2008.

[23] Nokia. Symbian OS. `http://symbian.nokia.com`.

[24] J. Oberheide. Android Hax. SummerCon 2010, June 2010. `http://jon.oberheide.org/files/summercon10-androidhax-jonoberheide.pdf` (accessed October 2012).

[25] J. Oberheide and F. Jahanian. When mobile is harder than fixed (and vice versa): demystifying security challenges in mobile environments. In *Proceedings of*

the 11th Workshop on Mobile Computing Systems and Applications, HotMobile '10, pages 43–48, New York, NY, USA, 2010. ACM.

[26] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in Android. In *Proceedings of the 25th Annual Computer Security Applications Conference*, ACSAC '09, pages 340 –349, dec. 2009.

[27] F. A. Petitcolas, R. J. Anderson, and M. G. Kuhn. Information hiding-a survey. *Proceedings of the IEEE*, 87(7):1062 –1078, July 1999.

[28] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos. Paranoid Android: versatile protection for smartphones. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 347–356, New York, NY, USA, 2010. ACM.

[29] R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, NDSS '11, pages 17–33, Feb. 2011.

[30] A. B. Shaffer, M. Auguston, C. E. Irvine, and T. E. Levin. A security domain model to assess software for exploitable covert channels. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '08, pages 45–56, New York, NY, USA, 2008. ACM.

[31] S. Smalley, NSA, and Trust Mechanisms (R2X). SEAndroid. `http://selinuxproject.org/page/SEAndroid` (accessed October 2012).

[32] The Lookout Blog. Lookout's privacy advisor protects your private information. `http://blog.mylookout.com/2010/11/lookout%E2%80%99s-privacy-advisor-protects-your-private-information/` (accessed October 2012).

[33] M. Tiwari, J. K. Oberg, X. Li, J. Valamehr, T. Levin, B. Hardekopf, R. Kastner, F. T. Chong, and T. Sherwood. Crafting a usable microkernel, processor, and I/O system with strict and provable information flow security. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 189–200, New York, NY, USA, 2011. ACM.

[34] C.-R. Tsai, V. D. Gligor, and C. S. Shandersekaran. On the identification of covert storage channels in secure systems. *IEEE Transactions on Software Engineering*, 16:569–580, June 1990.

[35] T. Vennon and D. Stroop. Threat analysis of the Android market. Technical report, GTC, June 2010. Smobile systems technical report, Available at `http://threatcenter.smobilesystems.com/wp-content/uploads/2010/06/Android-Market-Threat-Analysis-6-22-10-v1.pdf` (accessed October 2012).

[36] Z. Wang and R. B. Lee. Covert and side channels due to processor architecture. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, ACSAC '06, pages 473 –482, dec. 2006.

Enabling Trusted Scheduling in Embedded Systems

Ramya Jayaram Masti[†], Claudio Marforio[†], Aanjhan Ranganathan[†],
Aurélien Francillon[‡], Srdjan Capkun[†]
[†]Institute of Information Security, ETH Zurich, Switzerland
[‡]Eurecom, Sophia-Antipolis, France
[†]{rmasti, maclaudi, raanjhan, capkuns}@inf.ethz.ch
[‡]aurelien.francillon@eurecom.fr

ABSTRACT

The growing complexity and increased networking of security and safety-critical systems expose them to the risk of adversarial compromise through remote attacks. These attacks can result in full system compromise, but often the attacker gains control only over some system components (e.g., a peripheral) and over some applications running on the system. We consider the latter scenario and focus on enabling on-schedule execution of critical applications that are running on a partially compromised system — we call this *trusted scheduling*. We identify the essential properties needed for the realization of a trusted scheduling system and we design an embedded system that achieves these properties. We show that our system protects not only against misbehaving applications but also against attacks by compromised peripherals. We evaluate the feasibility and performance of our system through a prototype implementation based on the AVR ATmega103 microcontroller.

1. INTRODUCTION

Today, security- and safety-critical systems are being increasingly networked to facilitate their remote configuration, control and monitoring. As a result, they face an increased risk of adversarial compromise and therefore have to be designed to meet their real-time constraints even if they are partially compromised. More specifically, it is necessary to architect them such that they can guarantee the execution of certain critical functionality despite the presence of other misbehaving system components (e.g., compromised applications, peripherals). We refer to this property of preventing applications and components under the attacker's control from changing the execution times of other applications as *trusted scheduling*. Recent examples of compromised embedded systems [10], control systems [16] and peripherals [15] show that this is an emerging problem.

Most safety-critical systems include a real-time operating system (RTOS) [5, 6] or similar system management software (e.g., microkernel [18]) whose primary goal is to ensure that their real-time constraints are met. More recently, these RTOS also include mechanisms to contain the effects of other misbehaving software components/applications. However, RTOS do not address threats by untrusted peripherals (e.g., an RTOS cannot prevent a compromised peripheral from making the peripheral bus unusable by not adhering to the bus protocol). Furthermore, their complexity makes them prone to vulnerabilities that can be exploited to force the system to deviate from its expected behavior [4].

In this work, we address the problem of enabling trusted scheduling in the context of security- and safety-critical embedded systems. These are specialized devices that typically include a CPU, memory and some peripherals connected to the CPU via the peripheral bus. They usually run a fixed set of applications whose resource requirements are well-known in advance. We first identify three essential components of a trusted scheduling architecture, namely, secure scheduling, secure resource allocation and application state protection. This is in contrast to conventional scheduling that only focuses on the CPU allocation.

Second, we describe an embedded system architecture that achieves trusted scheduling and analyze its security. Our architecture includes five main hardware components, namely, a scheduler which decides the order in which applications execute, time-slice and atomicity monitors that ensure CPU availability, an application-aware memory protection unit which mediates memory access and a peripheral bus manager which controls access to the peripheral bus. These components, together with a thin layer of software, ensure that misbehaving applications and peripherals cannot influence the system's expectation for other applications. We show that our architecture provides strong guarantees against remote attacks that exploit software vulnerabilities which we believe is crucial for today's safety-critical systems. We then evaluate the feasibility of realizing such an architecture through a prototype implementation based on the AVR ATmega103. Finally, we discuss how the design of system components (e.g., bus, peripherals) can affect the feasibility of achieving trusted scheduling on a particular architecture.

The rest of the paper is organized as follows. In Section 2 we discuss the problem of enabling trusted scheduling and identify the functions needed to achieve it in a system. In Section 3, we describe a trusted scheduling architecture for embedded systems and analyze its security. In Section 4, we discuss several practical security issues involved in realizing a trusted scheduling architecture. We present preliminary performance considerations in Section 4.2. Finally, we discuss related work in Section 5 and conclude in Section 6.

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

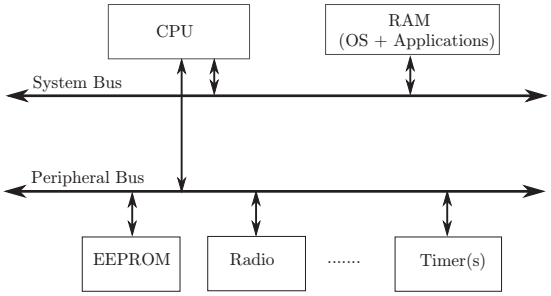


Figure 1: Most embedded systems consist of a *CPU*, *memory* (for code and data) and a set of *peripherals* that interact over a *system bus* and a *peripheral bus*. Typically, the peripherals are connected on a separate bus for efficiency reasons.

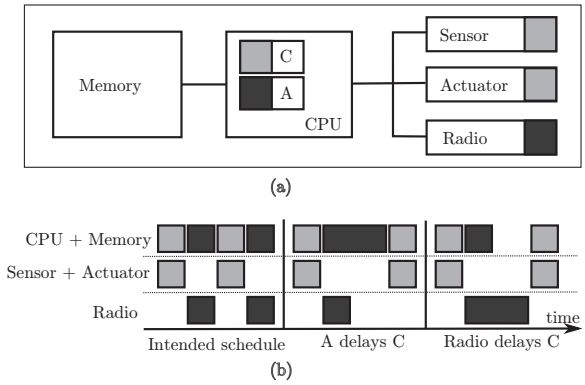


Figure 2: The execution schedule of a critical application *C* can be disrupted by a misbehaving (non-critical) application *A* that does not release the CPU on-time or exhausts the memory or by the *radio* peripheral that does not release the bus on-time.

2. TRUSTED SCHEDULING

In this section, we define trusted scheduling and identify the requirements that a system must satisfy in order to enable trusted scheduling.

2.1 Problem Statement

Trusted scheduling is the property of ensuring adherence to the intended execution schedule in adversarial settings.

To illustrate the problem of enabling trusted scheduling, we consider a simple embedded system $\mathcal{S}$ (Figure 1) that consists of a CPU, memory unit(s) and peripherals that are connected over one or more buses. Most embedded systems have two such buses: a system bus that is used to connect main components (e.g., CPU) to the memory unit(s) and a slower peripheral bus (e.g., SPI [9], I^2C [12]) that is used to connect the CPU to the peripherals (e.g., EEPROM, real-time clock, radio and other I/O interfaces). The system hosts a number of applications that are entirely independent and self-contained.

We consider an attacker $\mathcal{M}$ who controls a subset of applications and system components and is interested in interfering with the execution schedule of other (critical) applications that it does not control. For example, $\mathcal{M}$ could compromise an application and use it to gain control over the network card on the peripheral bus. We assume that $\mathcal{M}$ does not have physical access to the system and hence cannot launch physical attacks. We further assume that the attacker cannot influence any external inputs that affect the system's execution schedule.

This model corresponds to systems where critical and non-critical applications/peripherals co-exist. For example, a system can consist of a critical control (sensing/actuating) application and a non-critical communication application used for the sole purpose of reporting (Figure 2(a)). In this example system, if the radio peripheral or status reporting application is compromised, they could attempt to influence the execution schedule of the critical control application; this is illustrated in Figure 2(b). While conventional scheduling (or CPU scheduling) suffices to guarantee adherence to the intended schedule as long as all applications and peripherals are benign, it alone cannot provide similar guarantees in the presence of compromised applications and peripherals. Hence, in this work we focus on a broader notion of scheduling in adversarial contexts that we call trusted scheduling and define below.

We say that *a system enforces trusted scheduling if it prevents (possibly colluding) applications and components under the attacker's control from changing the execution times of other applications such that they do not correspond to the intended schedule.* In this work, we primarily focus on attacks that aim to delay or entirely prevent the execution of applications. We further assume that the applications do not fully depend on attacker-controlled applications or components for their execution; otherwise, little can be done to ensure trusted scheduling. Similarly, we also assume that the data influencing the execution schedule of applications either directly (e.g., as an input to the scheduler) or indirectly (e.g., persistent system data) cannot be modified by an attacker.

The problem that we want to solve in this work is that of designing an embedded system that enforces trusted scheduling, assuming that the attacker controls a subset of applications and system components. Existing real-time systems include software [6, 5, 18] and hardware [13, 19] to provide resilience against compromised applications, but do not consider misbehaving system components (peripherals).

2.2 Realizing Trusted Scheduling

Although systems are diverse and the application scenarios in which they are used largely differ, there are still some common functionalities that all systems must realize to support trusted scheduling which we discuss below.

To support trusted scheduling, a system should implement a robust scheduler, protect system resources and protect the applications. More precisely, the system must be designed such that the attacker (i) cannot modify the execution schedule of applications (ii) cannot interfere with the allocation of resources to the applications and (iii) cannot modify the state of applications (code and data).

This effectively implies that, a trusted scheduling system should implement a secure scheduler that schedules the execution of applications and enforces adherence to this schedule. Furthermore, this system should securely isolate appli-

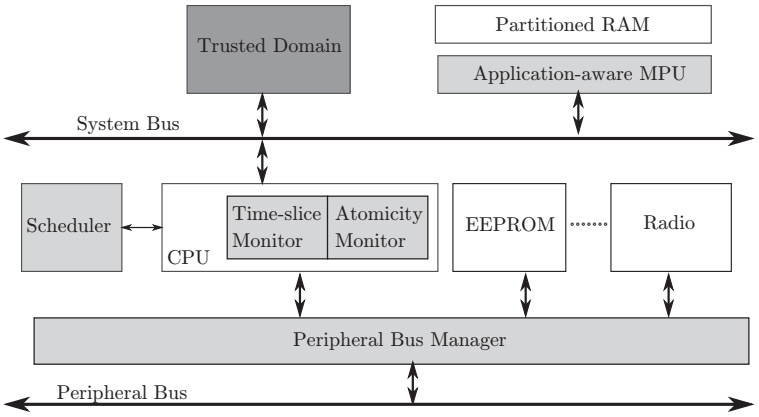


Figure 3: Our trusted scheduling architecture uses a thin layer of software (trusted domain) to initialize and configure its hardware components. The scheduler informs the trusted domain when a new application has to be executed. The trusted domain saves the current system state and transfers control to the new application. The CPU time-slice monitor and the atomicity monitor together guarantee that the trusted domain can regain control of the CPU when needed. Each application is allocated its own dedicated memory partition and application boundaries are enforced using an application-aware memory protection unit (MPU). Finally, the peripheral bus manager controls access to peripherals and also prevents misbehaving peripherals from denying applications access to the bus.

cations such that their code and data are protected, preventing the attacker from modifying applications or preventing their correct execution by modifying their data. Finally, the system should be able to securely multiplex shared system resources and ensure that applications are able to obtain all resources they need on-time for their correct execution. Ensuring the correct allocation of resources prevents internal Denial-of-Service (iDoS) attacks in which an attacker holds on to a system resource (e.g., a bus or a peripheral) required by another application or exhausts system resources to prevent other applications from running (e.g., dynamically allocating memory until it runs out).

In the next section, we describe our design of an embedded system that follows the above reasoning and supports trusted scheduling.

3. TRUSTED SCHEDULING SYSTEM

Our trusted scheduling system is designed for real-time systems with a pre-defined set of applications. It is tailored for embedded devices that are used in security- and safety-critical applications that have a well-defined and focused purpose. The system is initialized and configured by a trusted administrator and is not expected to be modified by the user during use.

3.1 System Overview

Our system is shown in Figure 3. It includes standard embedded system components (shown in white): CPU, RAM, system bus, peripheral bus, peripherals and trusted scheduling extensions (shown in gray): trusted domain, scheduler, application-aware MPU, time-slice and atomicity monitors and a peripheral bus manager.

The scheduler manages the execution of applications and informs the trusted domain when a new application must be executed. The scheduler triggers execution of applications according to its pre-determined schedule or in response to external events. The scheduler is therefore configured with the scheduling policy it must enforce and is aware of the applications that are running on the system. When a new application is scheduled to be executed, the trusted domain saves the current state of the system and transfers control to the new application. Before the actual execution of the application itself, the trusted domain configures and activates the desired CPU time-slice monitor (e.g., a timer interrupt) and the atomicity monitor that transfer control back to it from the application. The CPU time-slice monitor and the atomicity monitor together guarantee that the trusted domain can regain (with minimal latency) control of the CPU when needed. The scheduler is isolated from the applications and other system components; it acts directly on the CPU and thus can always stop and start the execution of the applications. In order to ensure application isolation, each application running on the system is allocated its own dedicated memory partition and application boundaries are enforced using an application-aware memory protection unit (MPU). Finally, to ensure that bus access to the peripherals is securely mediated, we introduce a peripheral bus manager. This peripheral bus manager controls access to the peripheral bus from the various peripherals and prevents misbehaving peripherals from denying the CPU (the running applications) and other benign peripherals access to the bus. It also ensures that each application can only access the set of peripherals to which it has been granted access by the trusted domain.

Every application starts executing at its first instruction and continues until it terminates, violates a security policy or is preempted. A security exception is raised if an application tries to access or modify code or data that does not belong to it. A security exception is also raised if an application tries to execute an atomic section that is larger than the preset limit or if its allocated CPU time-slice expires. Additionally, the peripheral bus manager also raises a secu-

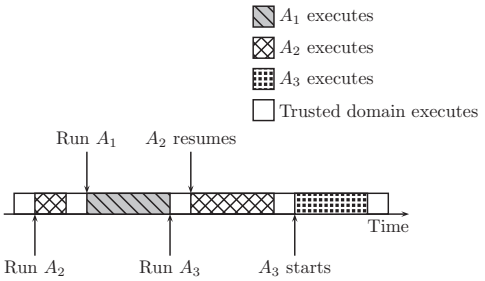


Figure 4: The trusted domain must save and restore state during a context switch to prevent applications from accessing and/or modifying each others state.

rity exception if any application tries to access a resource to which it has not been granted access or if the length of any bus transaction exceeds a pre-determined upper bound. Alternatively, the upper bound on the length of bus transactions can also be enforced by the atomicity monitor. When an application is preempted or forcibly terminated due to a security exception, control is transferred back to the trusted domain (via the hardware scheduler) which resumes execution of pending applications.

In terms of the system initialization, we assume that a trusted administrator supplies system initialization values to the trusted domain. When the system is powered-on, the trusted domain executes first and performs system initialization based on the inputs provided by the administrator. These inputs may include (but are not limited to) the number of applications, the peripherals they use, their memory layout, execution schedule, preemption policies (like the maximum CPU time-slice for each application, etc.). The trusted domain configures the scheduler for periodic and/or event-driven execution of applications. Furthermore, it also configures the resource allocator's CPU preemption policies and the application-aware MPU to create dedicated program and data partitions for each application. Finally, the trusted domain initializes the program memory of individual applications and configures the peripheral bus manager with information regarding the peripheral access requirements of the applications.

3.2 System Components

We now describe selected hardware and software components of the trusted computing base (TCB) in more detail.

Trusted Domain

The trusted domain is the only software component of the TCB and resides in ROM. The trusted domain is responsible for initializing the hardware scheduler, components that allocate system resources (described below), handling context switches between applications and the actual transfer of control from the scheduler to the individual applications (Figure 4). We chose this approach in order to prevent malicious applications from accessing or modifying the state of their predecessors which could result in a violation of trusted scheduling. The trusted domain uses a dedicated data partition in RAM that is not accessible by any other application or system component.

Scheduler

Our architecture includes a hardware-scheduler that uses information about external events and internal logic to decide the order in which applications execute. Although using a hardware scheduler limits flexibility (adding, removing applications, changing algorithms, etc.), we believe that this is an acceptable trade-off for security in real-time systems, as they are mostly single-purpose and run applications that do not change frequently (e.g., due to safety compliance issues). The scheduler transfers control to the trusted domain that in turn transfers control to the actual application.

Our scheduler implements *preemptive scheduling*. Although co-operative scheduling, where applications are allowed to execute to completion or co-operatively release the CPU, may be simpler and more efficient than preemptive scheduling, relying upon applications to release the CPU punctually can be dangerous in adversarial settings. This is because malicious applications may hold onto the CPU and hence delay (or in the worst case prevent) the execution of other applications. Hence, in order to achieve trusted scheduling, one must use preemptive scheduling schemes where the execution of any application can be interrupted at any point of execution except for those that the application explicitly labels as *atomic*.

Resource Allocation Components

The resource allocator must ensure that applications have access to at least three system resources while executing: CPU, memory and the bus(es).

CPU Availability

Atomic sections are required to ensure the correctness of certain operations (e.g., updating the stack pointer (SP)). The sequence of instructions in an atomic section (*(i)* decrementing the existing SP value and *(ii)* re-storing the new value in SP) should be executed without interruption. Otherwise, a corrupt system state (SP) may result. Hence, if an application is executing an atomic operation, the scheduler would have to wait until it completes the operation before suspending it. Although it is recommended that atomic operations should be as short as possible, a malicious application could declare large portions of code as atomic and hence delay, or in the worst case prevent the execution of another application. In order to prevent such attacks, the system must be able to bound the maximum duration (in time) of atomic operations.

Our architecture (Figure 3) includes an atomicity monitor that tracks the length of atomic code sections. The atomicity monitor terminates an erring application which exceeds a pre-defined execution upper bound. This requires that all applications are designed to respect this bound; otherwise they will fail to execute correctly. We note that this bound does not apply to the trusted domain. The atomicity monitor must also prevent nesting of atomic sections to increase their effective length. Furthermore, the trusted domain configures an additional CPU time-slice monitor (e.g., a timer) just before it transfers control to the individual application.

Memory Availability

Applications typically need two types of memory: program memory for their code and data memory for their stack and heap.

The use of shared stacks and heaps allows compromised applications to launch iDoS attacks by potentially exhausting stack or heap space. Recovery from such stack and heap overflow attacks requires invalidating current stack frames (e.g., by unwinding) and heap data. Although solutions that guarantee such secure stack sharing exist [24, 20], recovering from security violations can be complicated and expensive. This is because identifying and invalidating data memory regions that caused the violation can be both costly and time-consuming. For example, upon a security exception, one may have to unwind multiple stack frames which is time consuming compared to simply swapping the stack pointer. Enforcing access control policies in systems with shared data memory can also be complicated because it requires maintaining ownership information on individual stack frames and heap blocks as described in [20]. Therefore, we use dedicated data partitions for each application and depend upon the application-aware MPU (described below) to prevent iDoS attacks on data memory. A similar approach is used to ensure availability of program memory. Although such partitioned memory architectures limit flexibility, we believe they are still suitable for use in trusted scheduling architectures for such specialized systems whose memory requirements are typically known in advance.

In practice, most embedded systems use a memory protection unit (MPU) or a memory management unit (MMU) for protecting each application's state against unauthorized access or modification. Typically, such a unit only enforces access control policies (read, write, execute) at run-time on segments or pages of memory, and the operating system is responsible for separating memory regions of different applications. Our trusted scheduling architecture relies on a specialized MPU that combines these two functions, i.e., an application-aware MPU that not only checks for the type of access to memory but also whether the entity initiating such an access has the appropriate privileges. For specialized embedded devices with relatively long lifetimes, enhanced MPUs and MMUs that are application-aware (e.g., ARM [1], NIOS II [7]) present a reasonable trade-off between flexibility and better security.

In our system, the trusted domain configures the application-aware MPU with information regarding the boundaries of different applications. Every memory access (both program and data memory) is then mediated by this MPU.

Mediation of the Bus Access

If an application cannot gain access to the memory or to the peripherals that it needs for its correct execution, it will fail to execute. A compromised system component with access to a bus can cause such failures by holding on to the bus — preventing any communication among other system components that are connected to the same bus. We call this attack an iDoS attack on the bus.

In our system, we do not consider DMA-capable peripherals and assume that all peripherals access memory only through the CPU. We assume that the atomicity monitor enforces the upper bound on the length of bus transactions. This in turn prevents iDoS attacks on the system bus. We discuss this further in the security analysis (Section 3.3).

In addition to components connected to the system bus, components on the peripheral bus are also often crucial for the operation of the system or of individual applications, e.g., EEPROMs containing system configuration data may need to be accessed in a timely manner, an alarm application needs to have access to the radio peripheral to transmit alarm messages. In order to ensure the availability of the peripheral bus, we propose a secure hardware bus manager that mediates bus access requests.

In our system, we consider a multi-master (multi-)slave bus (e.g., I^2C). In most multi-master bus architectures, a typical data exchange consists of three phases: bus-arbitration, data-exchange and bus-release. Bus-arbitration is used to establish a bus-owner when there are several contenders (or masters). Arbitration is followed by data exchange and by an explicit bus-release phase in which other bus masters are informed about the availability of the bus. These three phases constitute a bus transaction and it is always executed atomically.

The shared nature of the multi-master bus allows a misbehaving peripheral to affect (by delaying or denying bus access) the execution of the applications that do not directly depend upon it. A misbehaving peripheral could deny bus access in the following ways:

(i) A misbehaving master peripheral may not respect the rules of bus arbitration and may continue its transmission beyond what it is allowed, thereby indirectly disrupting or modifying data on the bus.

(ii) A misbehaving master peripheral could gain access to the bus by sending carefully crafted data to win bus-arbitration every time and then sending data continuously without releasing the bus.

(iii) A misbehaving slave could delay its master node for arbitrary lengths of time.

In our system, we prevent these attacks by introducing a secure peripheral bus manager. This manager is configured by the trusted domain at start-up with information regarding the list of peripherals that must be accessed by each application. The manager uses this information at run-time to ensure that only peripherals that are needed by the executing application have access to the bus.

The manager further allows an application to selectively enable and disable peripherals at run-time. Such fine-grained control allows an application to only enable the peripheral that it is currently using. As a result, a misbehaving peripheral cannot influence any execution sequence during which it is not enabled and an application's interaction with any other peripheral that is not concurrently active. Together, these two features enable graceful degradation in the functionality of the application while maintaining tight security guarantees. However, little can be done if the misbehaving peripheral is critical for the application's correct execution.

Finally, the manager ensures that if an application terminates as a result of a security violation, then all the devices (peripherals) to which it had access are reset before they are re-used. We present a realization of a peripheral bus manager for the I^2C bus in Section 4.

3.3 Security Analysis

In this section, we analyze the security of our system. As described in Section 2, our attacker does not have physical access to the system, but can only remotely compromise selected applications and system components.

We assume that our trusted scheduling extensions (scheduler, trusted domain, application-aware MPU, peripheral

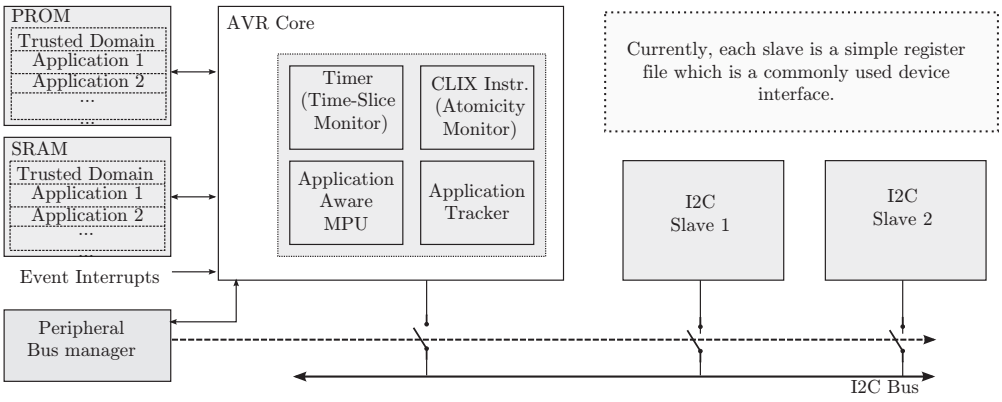


Figure 5: Our prototype implementation consists of a modified AVR core, partitioned data and program memories and a peripheral I^2C bus manager. The trusted domain also resides in its own dedicated RAM partition which other applications cannot access. The time-slice monitor is implemented using one of the AVR's timers and the atomicity-monitor is implemented as a custom instruction (`clix`). The application-aware MPU enforces memory boundaries of applications using information from the application tracker. The peripheral I^2C bus manager mediates access to the I^2C bus according to its access control map.

bus manager, time-slice/atomicity monitors) are implemented correctly and were initialized properly in a trusted initialization step that is free from adversarial interference. Furthermore, all of these components have simple logic, which reduces the risk of vulnerabilities in their implementation. We assume that the attacker cannot control inputs to the scheduler either directly (as external input) or indirectly (through applications that it controls). Allowing only the trusted domain to access the trusted scheduling components eliminates the risk of compromise of these secure elements by other applications. We assume that any system update (applications or configuration) involves a similar trusted initialization step and is adversary-free.

We analyze our system assuming that the attacker controls (up to) all of the applications on the system, except the critical application and (up to) all of the peripherals except the ones needed by the critical application. We show that even in this scenario, the execution schedule of the critical application will not be affected by the attacker. We further show that even if the attacker controls a subset of the peripherals that are used by the critical application, our system still ensures that the application executes on time, albeit with a reduced or compromised functionality due to the compromise of the peripheral.

Compromised Applications

The CPU time-slice monitor and the atomicity monitor ensure that misbehaving applications cannot indefinitely deny CPU access to other applications — no application can occupy the CPU longer than its assigned duration or execute bus transactions/atomic operations longer than a predefined length. This prevents iDoS attacks on the system bus by misbehaving applications since they can occupy the CPU (which is the only master of the system bus) only for a limited amount of time. Since an application can use a peripheral only as long as it has CPU-context, the time-slice monitor also prevents iDoS attacks against peripherals from misbehaving applications.

The application-aware MPU isolates applications such that they are restricted to using only their own code and data memory. This ensures that they do not occupy more than their share of memory space or interfere with other applications. Furthermore, the application-aware MPU and the trusted domain mediate context switches and prevent unauthorized cross-application state access and modifications.

The above countermeasures prevent one or more misbehaving applications from delaying or preventing the execution of the critical application.

Compromised Peripherals

Our system enforces that all peripherals can access memory only through the CPU (no DMA) and hence through the application-aware MPU; given this, there is no threat of iDoS attacks by peripherals on the system bus. However, a peripheral can attempt to disrupt trusted scheduling by launching iDoS attacks against the peripheral bus. This attack is prevented by the use of the peripheral bus manager that fully mediates the access of peripherals to the bus. The peripheral bus manager ensures that only the peripherals required by the currently executing application are active. This prevents other compromised peripherals from interfering with the execution of an application.

Furthermore, the system prevents peripherals from executing a long bus transaction — the length of any bus transaction (which is an atomic operation) is bounded by the atomicity monitor. Hence, misbehaving peripherals alone cannot disrupt trusted scheduling or the communication between the critical application and its peripherals.

Finally, even in the case of misbehaving applications and peripherals that are under the control of the same attacker, the combination of the above mechanisms ensures that the execution schedule of the critical application is not modified provided it does not depend on the misbehaving peripherals.

4. IMPLEMENTATION AND EVALUATION

In order to demonstrate the feasibility of realizing a trusted scheduling architecture, we implemented a prototype embedded system based on the AVR ATmega103 core. Our prototype (Figure 5) is a simplified instance of the architecture shown in Figure 3. It consists of a modified AVR core

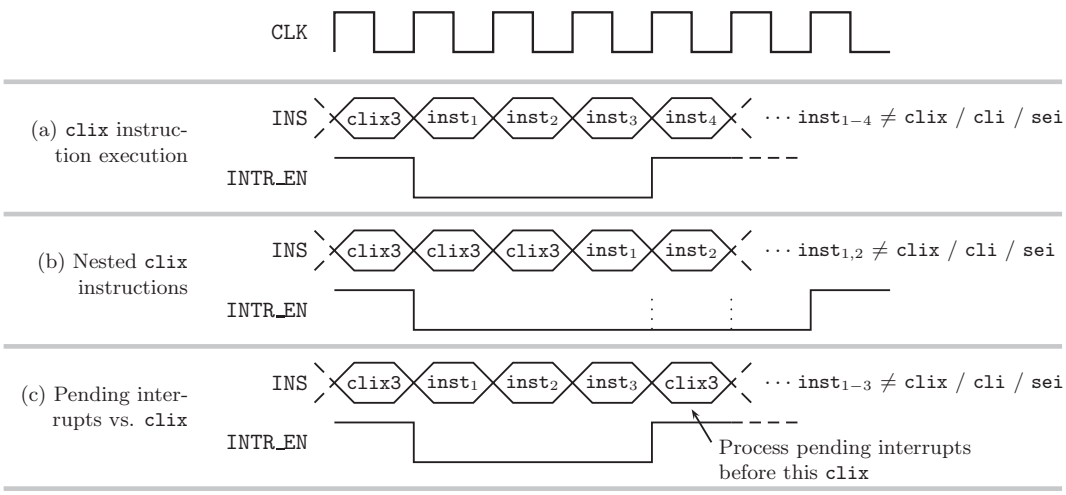


Figure 6: In order to limit the number of cycles for which interrupts can be disabled by untrusted applications, we introduce a new processor instruction (`clix`) which can be used to bound the maximum duration for which interrupts are disabled. We also implemented additional safeguards to prevent an application from disabling interrupts for a longer period by executing `clix` continuously or at regular intervals.

that is connected to two slave devices (register files) on an I^2C bus. We chose the AVR ATmega core and the I^2C bus due to their widespread use in embedded systems. In what follows, we describe our implementation in detail and then present initial results on the performance of our prototype.

4.1 Our Prototype

In our prototype, the trusted domain, which manages the scheduler, time-slice and atomicity monitors, application-aware MPU and the peripheral bus manager, is hosted in its own dedicated portion of RAM (instead of ROM) which is not accessible by other applications. The trusted domain also has exclusive access to a set of registers that hold security-critical information. This was necessary to ensure that misbehaving applications do not access or modify security-critical system parameters, e.g., the maximum length of an atomic section that is stored in a register and initialized during system start-up by the trusted domain. We added secure registers to store information regarding the currently executing application (in the application tracker), the trusted domain's stack pointer and the boundaries of different applications. We also modified the interrupt mask register (that is used to determine the set of active interrupts and hence, the set of applications that may interrupt the current application) and the timer corresponding to the time-slice monitor to be secure registers.

The scheduler is implemented using the set of prioritized interrupts available in the AVR core. It supports priority-based scheduling of applications with fixed and unique priorities. The highest priority interrupt line is reserved for the trusted domain and applications are executed as interrupt handlers. A high priority panic interrupt is fired when a security violation (as detected by the application-aware MPU, time-slice monitor, atomicity monitor or peripheral bus manager) occurs. It is important that the security panic interrupt is non-maskable, i.e., it can never be disabled. CPU preemption occurs automatically based on the priority of an incoming interrupt. The CPU time-slice monitor is implemented as a hardware timer that fires a security interrupt when it expires. The atomicity monitor is implemented as a custom instruction.

Furthermore, since the original AVR does not have an exclusive MMU or MPU module, we extend the core with a custom MPU that is initialized by the trusted domain. During system start-up the trusted domain loads the memory map of applications into the MPU. Then the MPU enforces application boundaries in program and data memory using the information about currently executing application that it obtains from the application tracker.

Memory partitions that are enforced by the MPU are created at compile time, i.e., by compiling the trusted domain and applications into a custom executable. The executable contains information regarding the load addresses of applications in RAM and their data regions (stack, heap and globals) and hence facilitates mapping of applications to separate (program and data) partitions. The trusted domain multiplexes the stack pointer between different applications. We extended the AVR core with an additional stack pointer that is used exclusively by the trusted domain. The trusted domain is also responsible for saving and restoring application contexts. We store the context of an interrupted application on the trusted domain's stack, which is accessible only by the trusted domain. This prevents any malicious application from accessing and modifying the state of any other application.

Atomicity Monitor

Since our prototype uses interrupt-based preemption mechanisms, applications disable all interrupts before executing atomic operations. In order to limit the maximum length of atomic operations, we added a custom instruction `clix Y` to the AVR core that disables all interrupts for `Y` cycles (Figure 6). We extended `gcc` (version 4.3.2) and `binutils` (version 2.20.1) to enable support for this new instruction. The value of `Y` is fixed at compile-time based on the atomic section's declared length. The generated application binaries use the custom instruction `clix` in place of the conventional `cli` instruction and a security exception is raised if any ap-

plication (other than the trusted domain) executes the `cli` instruction.

It is important to derive a practical bound on the maximum duration for which interrupts can remain deactivated by an untrusted application using `clix`. We refer to this upper bound as `max_clix`: the largest argument (maximum value for `Y`) that can be processed by a single `clix` instruction. This value is held in a dedicated register initialized by and accessible only to the trusted domain. Furthermore, enforcing the upper bound using `clix` requires the following additional safeguards:

(i) No application other than the trusted domain is allowed to execute the `cli` instruction.

(ii) Misbehaving applications may try to nest the execution of critical sections, i.e., they may execute the `clix` instruction consecutively and hence increase the effective number of cycles for which interrupts remain disabled (Figure 6). We prevent this in our prototype by ignoring `clix` instructions that occur while an older `clix` is being processed.

(iii) Pending interrupts should always be processed in precedence to `clix` instructions (Figure 6). This is important in the case when an interrupt occurs between a `clix a` and `clix b` instruction that are exactly `a` cycles apart. It must also be ensured that the `clix b` instruction is processed once the interrupt handler has completed.

Peripheral I²C Bus Manager

Our prototype includes a peripheral bus manager (Figure 7) that controls access to an I^2C bus that connects the CPU to peripherals. The I^2C bus is a two-wire serial bus. One of the wires is used for data and the other for control or clock signals. In practice, peripherals are connected to the bus using tri-state buffers (one each for the data and clock lines). Each peripheral typically controls its own tri-state buffer.

In our prototype (Figure 7), the access control map of the peripheral bus manager is initialized by the trusted domain with information regarding the set of peripherals that each application is allowed to access. Additionally, each application can choose to enable only a subset of all the peripherals to which it has been granted access using the peripheral selection register (PSR). The bus manager restricts access to the bus by controlling the enable signal of the tri-state buffer that connects the peripheral to the bus, i.e., the peripheral can access the bus only when the enable signal from the bus manager is also low. Finally, on the occurrence of a security violation (panic is high), the bus manager resets all the currently active devices so that they are ready for use by the next application that executes.

4.2 Preliminary Evaluation

In this section, we present an initial evaluation of our prototype with respect to its timing properties and hardware resource utilization.

Application Activation Latency

We evaluate the timing properties of our prototype in terms of its activation latency. Activation latency refers to the time elapsed between the arrival of a request for application

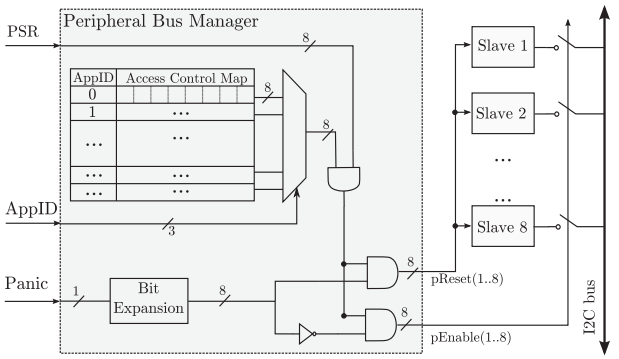


Figure 7: The secure I^2C bus manager (shown in gray above) restricts access to the bus by controlling tri-state buffers that are used to connect peripherals to the I^2C bus. The trusted domain initializes the peripheral access control map with information regarding the peripherals which each application can access. Each application can in turn enable a subset of these peripherals using the Peripheral Select Register (PSR).

execution and the actual execution of the application. When the system is idle, the activation latency is equal to the context switch time and we call this the *ideal activation delay*. In our prototype, this includes time required to save the current program counter, all the general purpose registers (32 registers in the case of the AVR ATmega103), the current stack pointer, timer register representing the remaining execution time, peripheral selection register and the interrupt mask register. This takes about 114 CPU cycles ($\simeq$9.1 μs when the processor runs at 12.5 MHz) and is our system's ideal activation time.

In the non-idle state of the system, the activation latency depends on whether execution control rests within another application or a context switch. If execution control rests within an atomic section of a lower-priority application, then the activation delay increases by the number of remaining cycles in the atomic section. Similarly, if the system is currently executing a context switch, then the activation delay increases by the number of remaining cycles in the context switch. Our measurements on the prototype implementation resulted in a worst case activation latency of 222 CPU cycles ($\simeq$18 μs at 12.5 MHz).

Finally, we calculate the system recovery time, i.e., the time taken by the system to exit an erring application and begin restoring a previous context. Our prototype implementation took 12 CPU cycles ($\simeq$1 μs at 12.5 MHz) to recover from a security panic and restore last known stable system state.

Application Execution Latency

In our prototype, the only component that directly affects the total application execution time is the MPU, which checks every memory access to guarantee its validity. The ATmega 103 has two types of data memory access instructions: direct and indirect. While indirect memory access instructions (`load` and `store`) do not incur additional delay, the direct memory access instructions require one extra cycle (i.e., they take 3 cycles instead of 2). This is because in the latter case,

the actual memory address which is fetched during the second cycle needs to be validated before the actual load/store operation.

Hardware Complexity

We implemented the trusted scheduling hardware modules as an extension to the AVR ATmega103 core in VHDL and synthesized it for a Xilinx Virtex 5 FPGA. The application-aware MPU, the time-slice monitor and the atomicity monitor together occupied 34.7% more logic units compared to the original AVR core. A major portion of the increase (about 22%) is the application-aware MPU which contains the memory map of application boundaries. However, we note that many CPUs today are already equipped with process-aware or domain-aware MMUs (e.g., NIOS II, ARM) which can potentially be used to realize application-aware MMUs at no additional hardware cost. The logic units utilized by the peripheral bus manager was insignificant (less than 0.1% of the whole system).

4.3 Discussion

In this section, we discuss the implications of choosing a bus protocol for a trusted scheduling enabled embedded system. Broadly, bus protocols can be classified as either *(i)* node-oriented (e.g., I^2C) or *(ii)* message-oriented (e.g., CAN [2]). In a node-oriented protocol, only one master and slave are active at any point in time. A simple bus manager design would be to allow access to the bus based on the priority of the master node. In addition, as implemented in our prototype, the master node can selectively enable the slave(s) that it requires for functioning. However, in message-oriented protocols like CAN, bus arbitration depends upon the priority of the message being broadcasted. Since multiple nodes can send out messages of the same priority, it is non-trivial to formulate secure bus access control policies for message-oriented protocols without any modifications to the protocols. Therefore, bus manager designs for message-oriented protocols requires further exploration.

Furthermore, in case the bus protocol uses a bi-directional bus line, peripherals may either connect to it using a single bi-directional pin [11] or using separate pins for input and output. Bus isolation circuits that allow control of physical access to the bus are much simpler in the latter scenario because it is easier to identify when a peripheral is actually transmitting. We intend to investigate this and other aspects of bus-interface designs that affect trusted scheduling as future work.

5. RELATED WORK

Given the feasibility of compromising the firmware of peripherals [15], there have been efforts to detect [22] as well as defend applications [28] against such compromised devices. The work that comes closest to ours in terms of protection against malicious peripherals is CARMA [28], which relies on Cache-as-RAM mechanism to securely sandbox applications. However, CARMA focuses on reducing the trusted computing base rather than providing trusted scheduling guarantees as presented in this work. Furthermore, our work addresses iDoS attacks by peripherals unlike CARMA which addresses attacks against confidentiality and integrity of application code and data.

Although there has been no direct work that provides guarantees similar to those of our trusted scheduling architecture, individual components of the architecture have been explored extensively in previous work. We summarize previous work on CPU scheduling, memory management and bus isolation in real-time embedded systems.

Most previous work on scheduling in real-time systems focused on optimizing the design [23, 26] and implementation of schedulers in hardware and software [30]. Today, scheduling in real-time systems is usually done by a real-time operating system (RTOS) [5, 6] or a separation kernel [3, 8]. An overview of contemporary RTOS and their performance can be found in [21, 27]. Most RTOS support the use of both co-operative and preemptive scheduling using priority- and round-robin-based algorithms. However, unlike our solution, none of these works explicitly include mechanisms to limit the length of atomic sections in code.

Process/Domain-aware MMUs are available in some of today's processors (e.g., ARM [1], NIOS II [7]). While most RTOS support the use of MMUs and MPUs, it is unclear whether they also support use of such application-aware MMUs as described in this work. Furthermore, commercial RTOS [5, 6] assign separate program and data memory partitions to each of the applications [27]. RTOS for memory constrained embedded devices ensure more efficient use of memory by sharing stack, heap and global data sections. Solutions for secure stack sharing [20, 24] and stack overflow prevention in such constrained devices [14, 17, 20] also exist.

The work in [25, 31] discusses DoS attacks against the system bus and defense mechanisms in the context of shared-memory multi-processor systems. However, these works only consider attacks by misbehaving applications running on one or more CPUs and do not take into account other misbehaving system components and peripherals. The need for fault tolerant bus design has led to the design of bus isolation solutions [11, 29]. These bus isolation solutions by themselves only provide a mechanism to physically isolate faulting devices and therefore useful for trusted scheduling only when they are configured and controlled by a context-aware bus controller as described in our design.

6. CONCLUSION

In this work, we investigated the problem of enabling trusted scheduling on embedded systems in adversarial settings. First, we identified the essential properties of a trusted scheduling system and presented an embedded system design that satisfies these properties. Our design includes a software-based trusted domain that manages the other hardware components. We analyzed the security of our proposal and showed that it achieves trusted scheduling in the presence of not only misbehaving applications but also misbehaving peripherals. Our prototype implementation based on the AVR ATmega103 shows the feasibility of realizing such an architecture through simple hardware extensions.

Acknowledgments

The research leading to these results was supported, in part, by the Hasler foundation (project number: 09080) and European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n^o 258754.

7. REFERENCES

[1] ARM Architecture Reference Manual. http://infocenter.arm.com/help/index.jsp.

[2] CAN Specification v2.0. http://esd.cs.ucr.edu/webres/can20.pdf.

[3] Codezero. http://www.l4dev.org/.

[4] IBM X-Force 2010 Trend and Risk Report. ftp://public.dhe.ibm.com/common/ssi/ecm/en/wgl03003usen/WGL03003USEN.PDF.

[5] Integrity for Embedded Systems. http://www.ghs.com/products.html.

[6] Lynx Embedded RTOS. http://www.lynuxworks.com/rtos/rtos.php.

[7] NIOS II Processor Reference Handbox, chapter 3. http://www.altera.com/literature/lit-nio2.jsp.

[8] OKL4 Microvisor. http://www.ok-labs.com/products/overview.

[9] SPI BlockGuide v03.06. http://www.ee.nmt.edu/~teare/ee308l/datasheets/S12SPIV3.pdf, 2003.

[10] Attacks on Mobile and Embedded Systems: Current Trends. https://mocana.com/pdfs/attacktrends_wp.pdf, 2009.

[11] Designing an isolated I2C Bus interface by using digital isolators. http://www.ti.com/lit/an/slyt403/slyt403.pdf, 2011.

[12] I^2C bus specification and user manual. http://www.nxp.com/documents/user_manual/UM10204.pdf, 2012.

[13] J. Adomat, J. Furunas, L. Lindh, and J. Starner. Real-time Kernel in Hardware RTU: A Step Towards Deterministic and High-performance Real-time Systems. In *Proceedings of the 8th Euromicro Workshop on Real-Time Systems*, pages 164 –168, 1996.

[14] S. Biswas, T. Carley, M. Simpson, B. Middha, and R. Barua. Memory Overflow Protection for Embedded Systems Using Run-time Checks, Reuse, and Compression. *ACM Transactions on Embedded Computing Systems*, 5(4):719–752, Nov. 2006.

[15] L. Duflot, Y.-A. Perez, and B. Morin. What If You Can't Trust Your Network Card? In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, RAID'11, pages 378–397, 2011.

[16] N. Felliere, L. O. Murchu, and E. Chien. W32.Stuxnet Dossier, 2011.

[17] A. Francillon, D. Perito, and C. Castelluccia. Defending Embedded Systems Against Control Flow Attacks. In *Proceedings of the 1st ACM Workshop on Secure execution of untrusted code*, SecuCode '09, pages 19–26, 2009.

[18] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 207–220, 2009.

[19] P. Kohout, B. Ganesh, and B. Jacob. Hardware Support for Real-time Operating Systems. In *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS'03, pages 45–51, 2003.

[20] R. Kumar, A. Singhania, A. Castner, E. Kohler, and M. Srivastava. A System for Coarse Grained Memory Protection in Tiny Embedded Processors. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 218–223, 2007.

[21] B. Leiner, M. Schlager, R. Obermaisser, and B. Huber. A Comparison of Partitioning Operating Systems for Integrated Systems. In *Computer Safety, Reliability, and Security*, volume 4680 of *Lecture Notes in Computer Science*, pages 342–355. 2007.

[22] Y. Li, J. M. McCune, and A. Perrig. VIPER: Verifying the Integrity of PERipherals' Firmware. In *Proceedings of the 18th ACM conference on Computer and Communications security*, CCS '11, pages 3–16, 2011.

[23] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, Jan. 1973.

[24] B. Middha, M. Simpson, and R. Barua. MTSS: Multitask Stack Sharing for Embedded Systems. *ACM Transactions on Embedded Computing Systems*, 7(4):46:1–46:37, Aug. 2008.

[25] T. Moscibroda and O. Mutlu. Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems. In *Proceedings of the 16th Usenix Security Symposium*, pages 257–274, 2007.

[26] K. Ramamritham and J. Stankovic. Scheduling Algorithms and Operating Systems Support for Real-time Systems. *In Proceedings of the IEEE*, 82(1):55 –67, Jan. 1994.

[27] S. Tan and B. Tran Nguyen. Survey and Performance Evaluation of Real-time Operating Systems (RTOS) for Small Microcontrollers. *Micro, IEEE*, (99), 2009.

[28] A. Vasudevan, J. M. McCune, J. Newsome, A. Perrig, and L. van Doorn. CARMA: A Hardware Tamper-Resistant Isolated Execution Environment on Commodity x86 Platforms. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security*, ASIACCS'12, 2012.

[29] N. Venkateswaran, S. Balaji, and V. Sridhar. Fault Tolerant Bus Architecture for Deep Submicron Based Processors. *SIGARCH Computer Architecture News*, 33(1):148–155, Mar. 2005.

[30] M. Vetromille, L. Ost, C. Marcon, C. Reif, and F. Hessel. RTOS Scheduler Implementation in Hardware and Software for Real Time Applications. In *17th IEEE International Workshop on Rapid System Prototyping*, pages 163 –168, 2006.

[31] D. H. Woo and H.-H. S. Lee. Analyzing Performance Vulnerability due to Resource Denial-of-Service Attack on Chip Multiprocessors. In *Proceedings of the Workshop on Chip Multiprocessor Memory Systems and Interconnects*, CMP-MSI'07, 2007.

TRESOR-HUNT: Attacking CPU-Bound Encryption

Erik-Oliver Blass
Northeastern University
blass@ccs.neu.edu

William Robertson
Northeastern University
wkr@ccs.neu.edu

ABSTRACT

Hard disk encryption is known to be vulnerable to a number of attacks that aim to directly extract cryptographic key material from system memory. Several approaches to preventing this class of attacks have been proposed, including TRESOR [18] and LOOPAMNESIA [25]. The common goal of these systems is to confine the encryption key and encryption process itself to the CPU, such that sensitive key material is never released into system memory where it could be accessed by a DMA attack.

In this work, we demonstrate that these systems are nevertheless vulnerable to such DMA attacks. Our attack, which we call TRESOR-Hunt, relies on the insight that DMA-capable adversaries are not restricted to simply reading physical memory, but can write arbitrary values to memory as well. TRESOR-Hunt leverages this insight to inject a ring 0 attack payload that extracts disk encryption keys from the CPU into the target system's memory, from which it can be retrieved using a normal DMA transfer.

Our implementation of this attack demonstrates that it can be constructed in a reliable and *OS-independent* manner that is applicable to any CPU-bound encryption technique, IA32-based system, and DMA-capable peripheral bus. Furthermore, it does not crash the target system or otherwise significantly compromise its integrity. Our evaluation supports the OS-independent nature of the attack, as well as its feasibility in real-world scenarios. Finally, we discuss several countermeasures that might be adopted to mitigate this attack and render CPU-bound encryption systems viable.

1. INTRODUCTION

Hard disk encryption is an increasingly popular set of techniques for preserving the confidentiality of persistent data. In such approaches, each block is encrypted before writing it to disk, and blocks are decrypted after reading them from disk. Disk encryption is completely transparent from the user's perspective, and virtually all major operating systems support this security mechanism—e.g., BitLocker [16] for Microsoft Windows, FileVault [1] for Mac OS X, and dmcrypt [4] for Linux. A similar functionality is provided by prominent third-party applications such as TrueCrypt [28] and PGP [26].

However, it has been shown previously that an adversary with physical access to a machine can circumvent disk encryption and access sensitive data. For instance, by attaching a malicious device to a running target machine, the adversary can perform a so-called DMA attack [5, 2, 13, 3, 20, 12].[1] Certain peripheral hardware busses—such as FireWire, Thunderbolt, or ExpressCard—give direct, unfettered access to a system's main memory. As such, the malicious device simply reads out the encryption key used to encrypt the hard disk using a DMA transfer. Knowing the secret key, the adversary can decrypt the hard disk and access data. Such DMA attacks are not only academic, but have already been seen in the real world [22]. As of today, there are password-recovery toolkits available that render DMA attacks accessible to everyone [21]. In conclusion, DMA attacks pose a major threat to any unattended machine.

To mitigate the problem of DMA attacks, recent work [18, 25] has suggested moving the encryption key from RAM to the CPU, which is inaccessible via DMA. Additionally, encryption is solely performed using CPU registers, thwarting any attempts to reveal sensitive key material using DMA transfers. We refer to cryptographic systems with this property as *CPU-bound*.

In this paper, we show that CPU-bound hard disk encryption is insecure as presented in prior work. We present a *novel*, *realistic*, and *concrete* attack, where an adversary with access to a DMA-capable hardware bus can access encryption keys of a CPU-bound encryption system. The critical observation underlying our work is that attackers are not only able to read from a system's memory, but are also able to write arbitrary code and data into memory. Using this capability, we demonstrate that an attacker can expose a CPU-bound encryption key by injecting a small piece of code into the operating system kernel. This code transfers the encryption key from the CPU into RAM, from which it can be accessed using a standard DMA transfer.

To summarize, our contributions are the following:

- We demonstrate that by leveraging the write capability

[1]DMA, or Direct Memory Access, refers to the capability of peripheral system hardware to transfer data to or from main memory without the involvement of the CPU. This feature is intended to improve system performance, but comes at the expense of centralized memory access enforcement.

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

of DMA transfers, an attacker can bypass the protections afforded by CPU-bound disk encryption systems such as TRESOR [18] and LOOPAMNESIA [25].

- We experimentally validate the feasibility of the attack by implementing it against TRESOR in an *OS-independent* way that only depends upon details of the IA32(e) architecture. The resulting attack is capable of circumventing TRESOR in a matter of seconds without crashing or otherwise significantly compromising the integrity of the target system.

  We note that while we concretely focus on TRESOR and FireWire-based DMA, our attack is directly applicable to all CPU-bound disk encryption systems, all IA32(e)-based systems, and all peripheral busses with DMA capabilities such as Thunderbolt or ExpressCard.

- We discuss potential mitigation strategies for our attack that improve the security of CPU-bound disk encryption.

The remainder of this paper is structured as follows. In Section 2, we present relevant background information on CPU-bound disk encryption, and on TRESOR in particular. We present our specific attack on TRESOR in Section 3, and evaluate its efficacy in Section 4. We discuss the feasibility of our attack and of CPU-bound encryption in Section 5. Finally, we present related work and briefly conclude in Sections 6 and 7.

2. BACKGROUND

CPU-bound disk encryption systems are intended to render normal disk encryption systems resilient to evil maid attacks [23], cold boot attacks [7], and other scenarios where attackers might gain physical access to a running target system. In this section, we describe the threat model, assumptions, and implementation of TRESOR [18], a recent, representative example of CPU-bound disk encryption. We stress, however, that while we ground our discussion in the example of TRESOR, the main ideas directly carry over to similar proposals such as LOOPAMNESIA. Where appropriate, we highlight details specific to TRESOR.

2.1 Threat Model

TRESOR adopts a strong adversarial model, in which attackers can execute arbitrary code in ring 3 on IA32 systems. Therefore, attackers can execute code with root privileges on UNIX-like systems, or as ADMINISTRATOR or SYSTEM on Windows-based systems. However, attackers should not be able to execute code in ring 0, and TRESOR takes several steps to prevent this from occurring—even when such execution would normally be allowed—that are described below. Similarly, attackers should not be able to access kernel memory.

TRESOR assumes that it is run directly on the hardware, and not as a virtualized guest that could be introspected by a privileged host operating system.

TRESOR does not guarantee that legitimate ring 0 code will not leak information about the encryption key from the CPU, e.g., by copying values from debug registers or executing program paths that are dependent on debug register values.

As TRESOR is intended to prevent most realistic physical attacks against hard disk encryption, attackers can examine the contents of memory using a number of techniques, including hardware bus inspection, cold boot attacks, and DMA transfers initiated by peripheral devices. It is assumed that directly inspecting CPU state is difficult, for instance by attaching JTAG debuggers or specialized hardware probes.

2.2 Implementation

TRESOR, and other CPU-bound disk encryption systems, maintains the confidentiality of the encryption key by confining it and the encryption process to the CPU. In the following, we discuss the specifics of how TRESOR accomplishes this.

Key Storage.

Because CPU registers are not directly accessible by DMA transfers, TRESOR uses them to store the encryption key. However, even though CPU registers cannot be directly accessed through DMA, they could be indirectly read without taking extra precautions. Software running in user space can be preempted, and user space registers will automatically be persisted to memory as part of a context switch. Using user space registers would therefore be prone against a well-timed DMA attack against the process control blocks of the kernel. This automatically disqualifies pure user space-based approaches.

Therefore, TRESOR uses the debug registers `dr0`, `dr1`, `dr2`, and `dr3`, giving a total of $4 \cdot 32 = 128$ bits of storage for 32-bit machines, or $4 \cdot 64 = 256$ bits of storage for 64-bit machines. This is enough to accommodate AES-128 or AES-256, respectively. In addition, the IA32 hardware specification ensures that the debug registers are inaccessible outside of ring 0 code.

An ASCII passphrase is initially entered at system boot time. The kernel then derives an AES key from the user's input using SHA-256 applied a number of times, and stores the resulting key into the debug registers. The kernel immediately wipes the memory used for the initial passphrase and AES key derivation. On multi-core machines, the same key is written in the debug registers of all cores.

Other CPU-bound encryption schemes use different registers for the same purpose. For instance, LOOPAMNESIA uses Intel Machine Specific Registers (MSRs), and the previous version of TRESOR uses the Intel Streaming SIMD Extension (SSE) registers [17].

AES Implementation.

TRESOR relies upon the Intel AES-NI instruction set to perform AES encryption. Recent Intel CPUs—e.g., Core i5 and Core i7—implement this instruction set to support AES encryption and decryption in hardware. The `aesenc(x,y)` CPU instruction performs one round of AES encryption (`SubBytes`, `ShiftRows`, `MixColumns`, and `AddRoundKey`) in one CPU cycle. Here, `(x,y)` are drawn from the set of sixteen SSE registers `xmm0` to `xmm15`. The first register `x` contains the current round key, and the second register `y` contains the current state of the AES encryption. The output of `aesenc`, the AES state, is written into the destination register `y`. Therewith, TRESOR encrypts one plaintext block completely outside of RAM.

To generate the different round keys completely outside of RAM, TRESOR uses the `aeskeygenassist` instruction. The

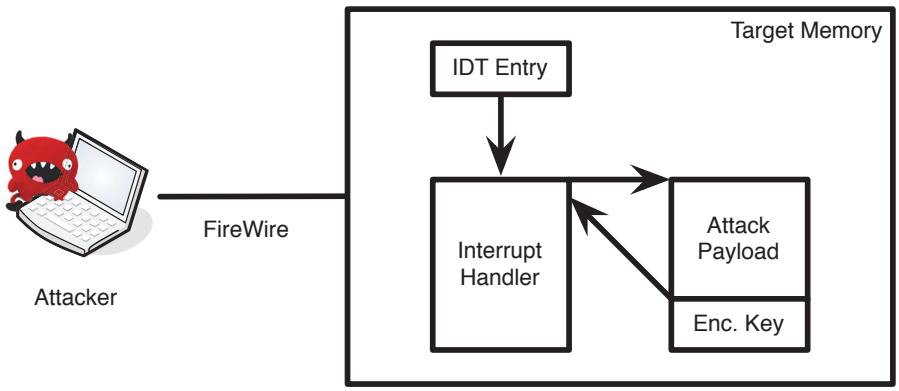


Figure 1: Overview of TRESOR-Hunt. The attacker overwrites physical memory in the target system to hook an interrupt handler. Then, an attack payload is executed that extracts the disk encryption key from the CPU into memory. The attacker can then initiate a DMA transfer to obtain the key.

AES key is copied from the debug registers into the first SSE register `xmm0`. Using `aeskeygenassist`, the 13 subsequent round keys can be derived from `xmm0` and are written into `xmm1` to `xmm13`.

To enable access to SSE registers during encryption, TRESOR defines the complete 14 round AES encryption within an atomic block in the kernel. This block cannot be interrupted, and SSE registers are saved while entering this block and restored before leaving. Being atomic, neither preemption by scheduling nor interrupts can disturb AES encryption and potentially leak the content of the registers to RAM.

Finally, TRESOR exports its AES interface to the Linux Crypto-API. This makes it available to standard Linux disk encryption software such as dm-crypt.

Kernel Integrity.

As dictated by the threat model, TRESOR assumes that attackers cannot run arbitrary code in ring 0. To enforce this, TRESOR takes additional steps to prevent the otherwise-allowed execution of attacker-controlled code in a kernel context.

1) The `ptrace` system call is modified to disallow access to the debug registers. Normally, `ptrace` allows user space programs such as debuggers to read and write debug registers, e.g., to set hardware breakpoints.

2) Kernel memory introspection from user space, enabled by special devices such as `/dev/kmem`, is disallowed. Otherwise, an attacker with sufficient ring 3 privileges could trivially read and write into kernel space, modifying, e.g., the atomic block within the kernel to read out debug registers.

3) Similarly, support for loadable kernel modules (LKMs) is removed. Otherwise, an attacker can insert a kernel module that reads out the debug registers.

These assumptions can be generalized to a *kernel integrity* property. Essentially, any CPU-bound encryption system depends on the integrity of kernel code, which is difficult to guarantee in general. We will now demonstrate that, even if the above assumptions hold and kernel integrity can be guaranteed against any local attacker with full user space privileges, DMA-based attacks are powerful enough to circumvent all of the above protection mechanisms.

3. A CPU-BOUND ENCRYPTION ATTACK

While CPU-bound disk encryption systems like TRESOR increase the difficulty for an adversary to compromise disk encryption keys, they are not impervious to attack under the threat model discussed in Section 2. In this section, we present an end-to-end attack against TRESOR, which we call TRESOR-Hunt. Note that while we focus on TRESOR as a case study, the techniques we employ generalize to any combination of CPU-bound disk encryption scheme, IA32-based OS, and DMA-capable hardware bus. In particular, while incorporating OS-specific knowledge would greatly simplify the attack—and would clearly be desirable should this information be confirmed for a particular target—we show that OS-specific details are not required, and that the attack is equally applicable to Windows, Mac OS X, and Linux without *a priori* knowledge of which OS is deployed on the target.

3.1 Attack Overview

Figure 1 presents a graphical overview of the attack. We outline the individual steps here.

1) First, the attacker gains physical access to the running IA32-based target system and attaches a device to a DMA-capable bus, e.g., FireWire.

2) The device initiates a DMA transfer to recover the contents of physical memory.

3) The device analyzes the physical memory dump, and identifies the kernel paging structures and interrupt descriptor table (IDT).

4) Using information derived from these structures, the device prepares an attack payload.

5) The device performs a DMA transfer to inject the attack payload into the memory of the target system.

6) The payload executes within a kernel context—i.e., ring 0—and copies the TRESOR encryption key from the CPU into a predetermined location in physical memory.

7) The device initiates a final DMA transfer to obtain the disk encryption key from memory.

Note that each of the above steps makes no assumptions on the target system aside from the presence of a DMA-capable bus and an IA32-derived architecture. In particular, virtually any modern OS for this architecture will make use of hardware paging for features such as process isolation and virtual memory, as well as interrupt handling for features such as device service requests and scheduling.

In the following, we discuss details of the individual steps of the attack.

3.2 Accessing System Memory

In our instantiation of the attack, we use a FireWire bus to perform DMA transfers to and from system memory. FireWire, or IEEE 1394, is a hardware specification for high-speed peer-to-peer device communication. Most importantly for our purposes, since FireWire is DMA-capable, it allows for unimpeded access to main memory.

Our prototype extends Inception [13], an existing tool for gaining privileged access to machines with an accessible FireWire bus. The tool's main purpose is to attack common (non-CPU-bound) disk encryption systems by using physical memory overwrites to grant root access to an attacker. The particular overwrites to perform are guided by a set of hardcoded signatures for a set of popular Linux-based operating systems. However, we merely build upon Inception's ability to read and write to physical memory. The remainder of our attack is built as a novel extension to the tool, and bears only superficial resemblance to its approach.

A notable limitation of FireWire is that it is limited to accessing the first 4 GB of memory. This limitation, however, did not impede the ability of our attack to successfully compromise TRESOR keys. This is due to the fact that the memory of interest resides at a relatively low physical address for the systems we attacked. Additionally, given the nature of the data structures we target, there is little reason to expect that this condition will not hold in other environments.

3.3 Hijacking Kernel Control Flow

Given a physical memory dump of the system obtained by Inception, the objective is then to analyze the dump in order to successfully execute code in ring 0—i.e., within the kernel context. An obvious approach here would be to use standard kernel hooking techniques; that is, to overwrite a known kernel function pointer to redirect control flow to code that we inject into physical memory. This approach, however, requires OS-specific knowledge to identify these function pointers and, therefore, is not suitable when the target OS is not known *a priori*.

Instead, the approach we adopt is to rely only upon data structures present in the target system's memory that are required by the IA32 architecture specification. In particular, our attack uses a combination of the kernel paging structures and interrupt descriptor table (IDT) to identify suitable locations to a) inject an attack payload to execute in ring 0, and b) to redirect control flow to that payload. In the following discussion, we refer to the IA32e—i.e., 64 bit—representation of these structures. Since it is relatively straightforward to heuristically infer the machine word size for a target architecture by examining a physical memory dump, we assert that this is without loss of generality.

IA32e Interrupt Descriptor Table (IDT).

The IDT is an IA32-specific structure that allows software to register handlers for system events such as interrupts and exceptions. The IDT itself is a contiguous array of descriptors that map an interrupt vector to an interrupt service routine (ISR). Each vector serves as an index into the array. Examples of standard interrupt vectors for the IA32 architecture include the breakpoint exception `#BP` (3), the general protection exception `#GP` (13), and page fault exception `#PF` (14). In addition, system-specific handlers can be mapped for vectors 32-255.

The CPU refers to the IDT's location in memory through the `IDTR` register, the value of which is loaded from memory and stored to memory using the `lidt` and `sidt` instructions, respectively. The `IDTR` specifies both the size of the IDT (minus 1), and the base of the table in memory.

The IDT serves as an ideal, OS-agnostic mechanism for locating code on IA32-based systems with the following properties: a) the code executes in ring 0, and b) it is potentially executed very often. While directly overwriting an IDT in memory is inadvisable due to the certain consequence that the machine will reset itself, each IDT entry does point to a function that can be hooked.

In particular, the approach we adopt is to select a system-specific interrupt vector, resolve its handler, and extract the first 16 bytes. We save these for later restoration. Then, we inject a jump to the location of our attack payload. This payload will be responsible for implementing our attack, as well as removing the hook and restoring the original initial ISR instruction sequence.

Identifying the IDT.

While the attack is, in principle, straightforward, there are several difficulties that arise. The first has to do with identifying the location of the IDT in memory. Recall that the standard means for accomplishing this is to execute the `sidt` instruction, which stores the value of the `IDTR` register into a specific location in memory. For instance, the following assembly routine would place the contents of `IDTR` in the memory location specified by the first argument in `rdi`.

```
; extern void __sidt(void *idtr)

bits 64
section .text

global __sidt

__sidt:
    sidt [rdi]
    ret
```

Unfortunately, this is a chicken-and-egg problem: to execute our attack, we need to locate the IDT, but before we gain control of the system, we cannot execute any instructions. Similarly, our DMA-based access to the system does not allow us to directly examine the state of the CPU.

Instead, we utilize a technique for heuristically identifying an IDT by scanning a physical memory dump. Our heuristics rely both on architectural constraints on IDTs as well as the fact that there exists much regularity in the values for each field of each IDT entry. While these constraints are certainly not foolproof, our experiments demonstrate that our heuristics are effective in practice.

In particular, our scan searches for a block of contiguous memory that satisfies the following properties.

1) The memory region is page-aligned to a 4 KB boundary.
2) The high-order bits of each entry's ISR are self-similar.
3) The type of each entry is one of the three permissible values—i.e., 5, 6, or 7.

We found that this approach was sufficient to reliably identify IDT locations in our experiments. For more details, please refer to Section 4.

Identifying IA32e Paging Structures.

Locating the IDT in memory is not the only challenge, unfortunately. A second obstacle arises from the fact that interrupt vector handlers are specified as linear addresses, while the attacker's device is restricted to a physical view of memory. That is, the target kernel is executing in protected mode, with a set of paging structures mapping linear addresses to physical addresses. In contrast, the attacker's device addresses physical memory. Therefore, without the ability to associate linear addresses with physical addresses, the device is unable to perform (at least) three critical tasks: a) locate the physical address of a particular ISR given its virtual addresses in the IDT, b) hook the ISR with the virtual address of the attack payload, and c) construct the attack payload such that it refers to the virtual address of the original ISR in order to remove the hook.

Therefore, in addition to locating the IDT, it is necessary to parse the kernel memory map as specified by the kernel paging structures. However, we face a similar situation to the case of the IDT and `sidt` instruction. The root of the paging structures is usually contained in the `cr3` register, but accessing that value requires execution of multiple `mov` instructions. Of course, the attacker is unable to do so at this point.

Accordingly, we adopt a similar approach of heuristically identifying the kernel's paging structures. Here, we rely on a combination of architectural constraints and OS-independent characteristics of kernel memory maps. In particular, we scan physical memory for a hierarchical paging structure that exhibits the following properties.

1) The paging structure tree is rooted at a valid page map level 4 (PML4) table.
2) PML4 entries, if present, point to valid page directory pointer tables (PDPTs).
3) PDPT entries, if present, point to valid page directories (PDs).
4) PD entries, if present, point to valid page tables (PTs).
5) Each node in the tree is page-aligned to a 4 KB boundary.
6) Reserved bits in each entry at each node of the tree are properly set to zero.
7) The ratio of pages with only ring 0 access to those with ring 3 access is above a fixed threshold.
8) The number of mapped pages is above a fixed threshold.

The first two properties above are architectural constraints; the second two are universal characteristics of kernel memory maps (most pages should only be accessible to the kernel, and a minimum number of pages should be mapped into physical memory). As in case of the IDT identification heuristics, the above was sufficient to uniquely identify the kernel paging structures in our experiments.

3.4 Preparing an Attack Payload

After resolving the location of the IDT and kernel paging structures, the next step is to construct the actual attack payload—i.e., the code that will be injected into the system to execute with ring 0 privileges. Given the address of the ISR to hook and the first 16 bytes of that ISR, this is quite simply accomplished by patching a compiled attack template such as shown in Figure 2. In particular, `INT_ADDR_MARK` is a special byte sequence that marks the location of the attack payload to patch with the address of the original ISR. Similarly, `INST_BUF_MARK` is a special byte sequence that marks the location to save the original initial instruction sequence for the target ISR.

3.5 Executing the Attack Payload

Execution of the attack payload requires two additional steps: a) injecting the payload into a writable and executable page in kernel memory, and b) patching the target ISR to redirect control flow to the location of the injected payload.

Our attack prototype accomplishes the first task by traversing the kernel memory map to discover a suitable physical page. The second task is completed by selecting a target ISR and replacing its initial instructions with a `jmp ATTACK_PAYLOAD_ADDR` instruction. At that point, the next time that the selected interrupt is raised, control of the system will be redirected to the attack payload. It will copy the contents of the debug registers to a predetermined location in RAM, unhook the targeted ISR, and continue execution of that ISR. The attacker's device will then copy out the encryption key, defeating the CPU-bound property of the disk encryption system.

4. EVALUATION

In the following section, we report on an evaluation of TRESOR-Hunt. In particular, we focus on two key aspects of the attack: a) how effective are the heuristics used to identify IDTs and kernel paging structures, and b) how effective is the attack in practice.

4.1 Data Structure Identification

The goal of this experiment is to quantify the accuracy of the heuristics we use to identify IDTs and kernel paging structures. Accordingly, we extracted physical memory dumps for several IA32-based operating systems, including Linux 3.3.7, FreeBSD 9.0, and Mac OS X 10.7.3, and applied our heuristics. In each case, we successfully identified

```
global extract_key

; extract the disk encryption key
extract_key:
    ; copy debug registers
    mov rax, dr0
    mov [dbg_regs.dr0 wrt rip], rax
    mov rax, dr1
    mov [dbg_regs.dr1 wrt rip], rax
    mov rax, dr2
    mov [dbg_regs.dr2 wrt rip], rax
    mov rax, dr3
    mov [dbg_regs.dr3 wrt rip], rax

    ; restore original instructions
    mov rdx, INT_ADDR_MARK
    mov rax, [inst_buf.x0 wrt rip]
    mov [rdx], rax
    mov rax, [inst_buf.x1 wrt rip]
    mov [rdx+0x08], rax

    ; jump to original handler
    jmp [rdx]

; original instruction buffer
inst_buf:
    .x0 dq INST_BUF_MARK
    .x1 dq 0x00

; debug register dump
dbg_regs:
    .dr0 dq 0x00
    .dr1 dq 0x00
    .dr2 dq 0x00
    .dr3 dq 0x00

global attack_len
attack_len dq $-extract_key
```

Figure 2: Example TRESOR-Hunt attack payload template.

OS	Handler MSBs
Linux 3.3.7	0xffffffff81000000
FreeBSD 9.0	0xffffffff80b00000
Mac OS X 10.7.3	0xffffff80002d0000

Table 1: Common bits for IDT entry handler addresses for a selection of IA32-based operating systems.

the IDT for each operating system when compared to the ground truth of executing the `sidt` instruction.

Table 1 displays the results of one aspect of the heuristics, namely the check for common bits of potential ISR handler addresses. For each OS tested, it is clear that many of the most-significant bits are shared, and they clearly correspond to kernel linear addresses, which tend to be located high in virtual memory.

A similar experiment was performed for the heuristics to resolve the location of the kernel paging structures. In this case, kernel drivers were written to directly access `cr3` as appropriate, since (as opposed to the `sidt` instruction) access to `cr3` is architecturally restricted to ring 0 code. In all cases, our heuristics were able to uniquely identify the correct location of the PML4 table.

As a result, while it is certainly possible to falsely identify an IDT or kernel paging structure, we conclude that our heuristics are effective in practice. We speculate that this might be partially attributable to the fact that we perform our scan as a linear sweep from low to high physical addresses. Since the data structures of interest tend to be initialized early in the boot process, they also tend to reside in low physical memory and are therefore discovered before false positives might be encountered.

4.2 Performance

In this experiment, we validate that TRESOR-Hunt is able to efficient and successfully extract disk encryption keys from the CPU. Since TRESOR is only available for Linux, we performed this experiment solely for that OS, although we speculate that the results of this experiment would not differ significantly in other environments.

We performed 10 trials of the attack against a Linux 3.0.31 kernel patched with TRESOR on machine with an Intel Core i7 CPU and 16 GB of memory. In each case, the time to perform the attack was dominated by the time to extract the initial physical memory dump, which was on the order of several minutes. In comparison, the time required to analyze the memory dump, construct the attack payload, inject the payload, and extract the key was negligible.

As a result, we conclude that the attack is highly feasible for the threat model described in Section 2, where an adversary can gain unobserved physical access to a DMA-capable bus on an unattended target machine.

5. DISCUSSION

We have demonstrated that CPU-bound encryption is insufficient to securely encrypt a hard disk in the face of DMA attacks. However, we will now discuss other techniques that allow efficient, realistic protection against such attacks.

Disabling DMA.

The most intuitive way to prevent DMA attacks is to simply disable DMA. Microsoft suggests this, for example to protect BitLocker full disk encryption [15]. Similarly, the "old" Linux FireWire protocol stack `ieee1394` (until 2010 [10]) offered a command line parameter `phys_dma=0` when loading the FireWire kernel module to completely disable DMA [8].

While disabling DMA is an effective protection of DMA attacks, this solution is not acceptable in many scenarios due to the implied performance penalty.

A more sophisticated way of disabling DMA for FireWire is used in Mac OS X Lion with FileVault 2 [1]: FireWire is only disabled is the host machine enters "sleep" mode, e.g., the lid of a laptop is closed. To re-enable DMA, the user password has to be entered on wake-up of the machine. This protection helps against DMA attacks, e.g., if a laptop is left somewhere "unattended", but does not offer any defense in case of desktop or server machines that are running most of the time unattended.

Device Whitelisting.

The DMA attack we have demonstrated requires attach-

ing a malicious device to a target machine's FireWire port. The malicious device initiates DMA transfers to carry out the attack. To do so, however, the target machine's FireWire controller has to generally enable DMA transfer for this device. By default, today's operating systems enable DMA for any device attached.

Consequently, one way to mitigate DMA attacks would be to *authenticate* any DMA device to the kernel before allowing to perform DMA transfer, i.e., enabling DMA at the FireWire controller. Only valid, benign FireWire devices trusted by the user will get permission to use DMA. However, the FireWire standard does not support device-host authentication. Enabling strong cryptographic authentication would require deep changes in the FireWire firmware on devices and host controllers.

Still, a weaker form of authentication, simple "identification" is possible even with standard FireWire devices and firmwares. Each FireWire device comes with a 64 bit Globally Unique IDentifier (GUID) readable by the kernel. As soon as a new FireWire device is attached to the FireWire bus, kernel and host controller reset and re-initialize the FireWire bus. As part of the initialization, the host controller can enable DMA for all attached devices.

For increased security, instead of simply enabling DMA by default, the kernel can check in advance the GUID of the attached FireWire device. If the GUID is part of a "whitelist" of allowed FireWire devices, the kernel will allow DMA for the host controller, otherwise the kernel will disable DMA. The only requirement for this is a kernel accessible whitelist where the user stores the GUID of his trusted FireWire devices. Using the FireWire stack of current Linux kernels, this whitelisting technique can be implemented in a straightforward manner by modifying the host controller initialization code in `init_ohci1394_dma.c` and `ohci.c`.

The above identification is clearly not authentication: if an attacker can spoof or guess the GUID of a trusted device, an impersonation attack is possible. Still, if the user protects access to his trusted FireWire devices, the GUID can provide sufficient protection against DMA attacks.

Hardware Disk Encryption.

With hardware disk encryption techniques, the actual encryption and decryption of data is performed by the hard disk itself and not by the host machine. Only during an initial phase at, e.g., boot time the host exchanges an encryption key with the hard disk. DMA attacks are impossible against hardware disk encryption. Many hard disk manufacturers offer hard disk with hardware disk encryption [24, 9, 27]. Although using hardware disk encryption successfully protects against DMA attacks, current solutions only fill a niche: they are (hardware) specific, only support the Windows operating system, often require support for Trusted Computing hardware on the host computer, and are expensive compared to software-based solutions.

IOMMU.

Similar to a traditional memory management unit (MMU), an IOMMU is a piece of hardware that controls access to physical memory for peripheral devices. Located between DMA-capable devices and the physical memory, it translates virtual addresses as used by DMA devices into physical memory addresses. Moreover, an IOMMU controls which device can read or write to which physical memory address.

Consequently, by using an IOMMU, the kernel could explicitly protect certain memory regions against reading or writing. The availability of an IOMMU would not only protect against DMA-based attacks, but also render CPU-bound encryption superfluous. As memory regions containing a cryptographic key can be protected using the IOMMU, there would be no need to perform encryption outside RAM anymore.

In practice, only recently Intel and AMD have introduced IOMMUs ("VT-d" and "AMD-Vi") for their latest chipsets. Currently IOMMUs are used in hypervisors to allow safe DMA transfers between attached devices and guest operating systems. As a matter of fact, today none of the popular operating systems supports IOMMU, and enabling support for IOMMUs requires significant changes to the operating system.

6. RELATED WORK

TreVisor [19] is a CPU-bound encryption system that isolates the encryption process in a hypervisor (BitVisor) on top of Linux. In addition, TreVisor uses Intel's VT-d IOMMU technology to restrict memory regions accessible by DMA to protect the integrity of the hypervisor. While effective, this approach has several drawbacks that render it impractical in the real world. First, the use of a hypervisor automatically disables virtualization software such VirtualBox and VMware, as well as rendering debug registers inaccessible. Second, although the authors use Intel's AES-NI [11] technology to compute individual AES rounds in hardware, performance decreases by up to 50% in this setup. Finally, Intel's recent VT-d IOMMU is not used by any major operating system today, and we do not conjecture its availability in the near future since using an IOMMU requires major changes to an operating system and its kernel.

Mac OS X's FileVault 2 [1] has been reported to disable FireWire and Thunderbolt DMA whenever the computer goes into "sleep mode" (standby), for example as soon as the user closes the lid of a laptop [14, 6]. If the user wakes up the computer from sleep, an "unlock password" has to be entered to resume normal operation and re-enable DMA. While in theory this is an effective countermeasure against any DMA attack, it is not applicable to a running system. Along the same lines, to protect BitLocker disk encryption again DMA attacks, Microsoft suggests disabling DMA transfer completely [15].

A related hardware attack against disk encryption systems is the cold boot attack [7]. A cold boot attack exploits the fact that contents of DRAM memory usually survives for some amount of time without power. An attacker can physically remove memory modules from a target machine, analyze them, and recover sensitive data. Contrary to DMA attacks, cold boot attacks can be mitigated with CPU-bound encryption.

7. CONCLUSIONS

CPU-bound encryption systems, such as LoopAmnesia and Tresor, attempt to prevent the disclosure of disk encryption keys from powerful adversaries that have full ring 3 privileges and physical access to the machine. In this paper, we present Tresor-Hunt, a *novel*, *realistic*, and *concrete* attack that bypasses the protection afforded by one such system.

Our attack relies on the insight that DMA-capable adversaries are not restricted to simply reading physical memory, but can write arbitrary values to memory as well. TRESOR-Hunt leverages this insight to inject a ring 0 attack payload that extracts disk encryption keys from the CPU into the target system's memory, from which it can be retrieved using a normal DMA transfer.

Our implementation of this attack demonstrates that it can be constructed in a reliable and *OS-independent* manner that is applicable to any CPU-bound encryption technique, IA32-based system, and DMA-capable peripheral bus. Furthermore, it does not crash the target system or otherwise significantly compromise its integrity. Our evaluation supports the OS-independent nature of the attack, as well as its feasibility in real-world scenarios. Finally, we discuss several countermeasures that might be adopted to mitigate this attack and render CPU-bound encryption systems viable.

8. REFERENCES

[1] Apple. FileVault 2. http://support.apple.com/kb/HT4790, 2012.

[2] B. Böck. Firewire-based Physical Security Attacks on Windows 7, EFS and BitLocker. http://www.securityresearch.at/publications/windows7_firewire_physical_attacks.pdf, 2009.

[3] A. Boileau. Hit by a Bus: Physical Access Attacks with Firewire. Ruxcon, http://www.security-assessment.com/files/presentations/ab_firewire_rux2k6-final.pdf, 2006.

[4] dm-crypt developers. dm-crypt: a device-mapper crypto target. http://www.saout.de/misc/dm-crypt/, 2012.

[5] M. Dornseif. Owned by an iPod: Firewire/1394 Issues. PacSec, http://md.hudora.de/presentations/firewire/PacSec2004.pdf, 2004.

[6] T. Garrison. Firewire Attacks Against Mac OS Lion FileVault 2 Encryption. http://www.frameloss.org/, 2011.

[7] J. Halderman, S. Schoen, N. Heninger, W. Clarkson, W. Paul, J. Calandrino, A. Feldman, J.Appelbaum, and E. Felten. Lest We Remember: Cold Boot Attacks on Encryption Keys. In *Proceedings of USENIX Security Symposium*, pages 45–60, San Jose, CA USA, 2008. USENIX Association.

[8] U. Hermann. Physical memory attacks via Firewire/DMA – Part 1: Overview and Mitigation (Update) . http://www.hermann-uwe.de/blog/physical-memory-attacks-via-firewire-dma-part-1-overview-and-mitigation, 2008.

[9] Hitachi. Safeguarding Your Data with Hitachi Bulk Data Encryption. http://www.hgst.com/tech/techlib.nsf/techdocs/74D8260832F2F75E862572D7004AE077, 2008.

[10] IEEE 1394 FireWire WiKi. JuJu Migration. https://ieee1394.wiki.kernel.org/index.php/Juju_Migration, 2012.

[11] Intel. Intel Advanced Encryption Standard Instructions (AES-NI). http://www.intel.com/, 2010.

[12] N. P. Jr., T. Fraser, J. Molina, and W. Arbaugh. Copilot – a Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceedings of USENIX Security Symposium*, pages 179–194, San Diego, USA, 2004.

[13] C. Maartmann-Moe. Inception. http://www.breaknenter.org/projects/inception/, 2011.

[14] C. Maartmann-Moe. Adventures with Daisy in Thunderbolt-DMA-land: Hacking Macs through the Thunderbolt interface. http://www.breaknenter.org/, 2012.

[15] Microsoft. Blocking the SBP-2 driver to reduce 1394 DMA threats to BitLocker. http://support.microsoft.com/kb/2516445, 2011.

[16] Microsoft. BitLocker Drive Encryption Overview. http://technet.microsoft.com/en-us/library/cc732774.aspx, 2012.

[17] T. Müller, A. Dewald, and F. Freiling. A Cold-Boot Resistant Implementation of AES. In *Proceedings of European Workshop on System Security*, Paris, FR, 2010.

[18] T. Müller, F. Freiling, and A. Dewald. TRESOR runs encryption securely outside RAM. In *Proceedings of USENIX Security Symposium*, San Francisco, CA USA, 2011. USENIX Association.

[19] T. Müller, B. Taubmann, and F. Freiling. TreVisor – OS-Independent Software-Based Full Disk Encryption Secure Against Main Memory Attacks. In *Proceedings of the Conference on Applied Cryptography and Network Security*, Singapore, 2012. To appear.

[20] P. Panholzer. Physical Security Attacks on Windows Vista. https://www.sec-consult.com/files/Vista_Physical_Attacks.pdf, 2008.

[21] Passware. Passware Kit Forensic 11.7. http://www.lostpassword.com/kit-forensic.htm, 2012.

[22] K. Poulsen. *Kingpin: How One Hacker Took Over the Billion-Dollar Cybercrime Underground*. Crown, 2011.

[23] B. Schneier. Evil Maid Attacks on Encrypted Hard Drives. http://www.schneier.com/blog/archives/2009/10/evil_maid_attac.html, October 2009.

[24] Seagate. Momentus 5400 FDE.2. http://www.seagate.com/docs/pdf/marketing/po_momentus_5400_fde.pdf, 2008.

[25] P. Simmons. Security Through Amnesia: A Software-Based Solution to the Cold Boot Attack on Disk Encryption. In *Proceedings of Annual Computer Security Applications Conference*, pages 73–82, Orlando, FL USA, 2011.

[26] Symantec. PGP Whole Disk Encryption. http://www.symantec.com/whole-disk-encryption, 2012.

[27] Toshiba. Toshiba Debuts Self-Encrypting Drive Technology. http://www.prnewswire.com/news-releases/toshiba-debuts-self-encrypting-drive-technology-at-rsa-conference-2009-61822062.html, 2009.

[28] TrueCrypt. Free Open-Source On-the-Fly Encryption. http://www.truecrypt.org/, 2012.

When Hardware Meets Software: A Bulletproof Solution to Forensic Memory Acquisition

Alessandro Reina
Dipartimento di Informatica
Università degli Studi di Milano

Aristide Fattori
Dipartimento di Informatica
Università degli Studi di Milano

Fabio Pagani
Dipartimento di Informatica
Università degli Studi di Milano

Lorenzo Cavallaro
Information Security Group
Royal Holloway, University of London

Danilo Bruschi
Dipartimento di Informatica
Università degli Studi di Milano

ABSTRACT

The acquisition of volatile memory of running systems has become a prominent and essential procedure in digital forensic analysis and incident responses. In fact, unencrypted passwords, cryptographic material, text fragments and latest-generation malware may easily be protected as encrypted blobs on persistent storage, while living seamlessly in the volatile memory of a running system. Likewise, systems' run-time information, such as open network connections, open files and running processes, are by definition live entities that can only be observed by examining the volatile memory of a running system. In this context, tampering of volatile data while an acquisition is in progress or during transfer to an external trusted entity is an ongoing issue as it may irremediably invalidate the collected evidence.

To overcome such issues, we present SMMDumper, a novel technique to perform *atomic* acquisitions of volatile memory of running systems. SMMDumper is implemented as an x86 firmware, which leverages the System Management Mode of Intel CPUs to create a *complete* and *reliable* snapshot of the state of the system that, with a minimal hardware support, is resilient to malware attacks. To the best of our knowledge, SMMDumper is the first technique that is able to atomically acquire the whole volatile memory, overcoming the SMM-imposed 4GB barrier while providing integrity guarantees and running on commodity systems.

Experimental results show that the time SMMDumper requires to acquire and transfer 6GB of physical memory of a running system is reasonable to allow for a real-world adoption in digital forensic analyses and incident responses.

Categories and Subject Descriptors

D.4 [**Operating System**]: Security and Protection — System Program and Utilities — Invasive software (e.g., viruses, worms, Trojan horses)

General Terms

Security

Keywords

Forensic, System Management Mode, Live Memory Acquisition

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

1. INTRODUCTION

Memory acquisition and analysis are well-known and long-term studied digital investigation activities. Although historically focused on information stored and likely hidden on persistent media (e.g., hard disks), digital investigation efforts have nowadays embraced the realm of volatile storage too. For instance, unencrypted passwords, cryptographic material, text fragments and latest-generation malware may easily be protected as encrypted blobs on persistent storage, while live seamlessly in the volatile memory of a running system. In addition, systems' run-time information, such as open network connections, open files and running processes, are by definition live entities that can only be observed on a live, running, system.

Given the importance of the problem and its related challenges, a number of solutions have been proposed to date. Overall, such approaches differ mostly by the degree in which the basic forensic requirements of *atomicity* (i.e., volatile acquisitions must occur within an uninterrupted time-frame), *reliability* (i.e., only trustworthy and consistent acquisitions must be retained), and *availability* (i.e., solutions must be device-agnostic)—informally described in [10]—are satisfied.

Roughly speaking, most solutions are either software- or hardware-based [19, 10]. The former are typically embedded into the operating system kernel, offering isolation (and thus protection) only against typical userspace attacks. Conversely, hardware-based solutions seem to be resilient against kernel-level threats too (for instance, Carrier and Grand rely on a DMA-based physical copy of the volatile memory built on top of a specific PCI device [2]).

Although perceived in principle to be more secure then their software counterparts, hardware-based solutions too may suffer from serious weaknesses, undermining their overall effectiveness. For instance, Rutkowska showed quite re-

cently how the privileges of a kernel-level malware can successfully manipulate specific hardware registers to provide device-dependent (i.e., CPU or DMA) split views of the contents of (volatile) memory [16]. As such, it is currently safe to assume that *pure* hardware-based state-of-the-art techniques are as robust (or weak) as their software-based counterparts, leaving memory acquisitions an open challenge.

To address such issues, one of the directions the research community is exploring revolves around the possibility of building dependable memory acquisition techniques on top of system management mode (SMM), a particular execution state of modern x86 CPUs.

Modern x86 CPUs enter SMM via system management interrupts (SMIs). Switching to SMM causes the CPU to save the current system execution state in the system management RAM (SMRAM), a private address space specifically set up for the purpose. Afterward, the execution flow passes to the SMI handler, which is resident in SMRAM. One important feature of the SMRAM is that it can be made inaccessible from other CPU operating modes. Therefore, it can act as a trusted storage, sealed from any device or even the CPU (while not in SMM). Although powerful, SMM has a critical issue: according to the Intel specifications, SMM can only access up to 4GB of physical memory, even on IA-32 CPUs that support physical address extension (PAE) [8], limiting the overall memory acquisition capability of such approaches.

Wang *et al.* propose an hardware-based approach to periodically trigger SMIs (and thus entering SMM) to read the RAM content and send it off to a remote server [21]. Although interesting, that approach has a number of limitations. First, it requires additional hardware (i.e., a PCI card) to be installed on the system *prior* to any live acquisition attempt. Second, no attempt to bypass the 4GB SMM-imposed memory upper bounds is made, greatly limiting the acquisition process, especially on modern hardware equipped nowadays with more than 4GB of physical memory. Finally, Wang *et al.*'s approach does not provide any integrity guarantee on the overall acquisition procedure.

To overcome such issues, we present SMMDumper, a novel technique to perform *atomic* acquisitions of volatile memory of running systems. SMMDumper is implemented as an x86 firmware, which leverages the SMM of Intel CPUs to create a *complete* and *reliable* snapshot of the state of the system that, with a minimal hardware support, is resilient to malware attacks.

Our solution is based on a *collector* and a *triggering* modules. The former is resident in SMRAM and is responsible for transmitting to a trusted host the entire processor state and the memory content, overcoming the SMM-imposed 4GB barrier when PAE is enabled, while providing integrity guarantees and running on commodity systems. Conversely, the triggering module takes care of activating the collector via SMIs. Ideally, a sound and bulletproof implementation of the triggering component should rely on hardware-based activation mechanisms (for instance, a common scenario may see a specific keystroke directly connected to the SMI CPU pin). These artifacts would isolate the protected software component, making it inaccessible from user- or kernel-space. Although such mechanisms have already been proposed in the literature (e.g., [9]), we have opted for a software-only proof-of-concept implementation of the triggering module, for simplicity (a software-emulated hardware triggering mechanism does not affect the effectiveness of SMMDumper).

Roughly speaking, our software-based SMI triggering solution works as follows. We modify the local vector table (LVT) of the APIC controller to trigger an SMI upon the pressure of an appropriate keystroke combination. Once the SMI is triggered, the CPU switches to SMM, the current system state is saved and our SMI handler (i.e., the collector module) is executed. Although as outlined above our current SMI triggering implementation is vulnerable to kernel-level threats (e.g., SMI invocation avoidance via LVT reconfiguration to reroute keyboard interrupts), SMMDumper's underlining idea remains sound and, moreover, its triggering module implementation can easily be extended to rely on hardware-based mechanisms (e.g., [9]).

In summary, we make the following contributions:

1. We devise a novel firmware-based technique to create a complete and reliable snapshot of the state of the system that, with a minimal hardware support, is resilient to any malware attack.

2. We devise an SMM-based mechanism that enables us to access any physical memory extending over 4GB.

3. We devise a mechanism to digitally sign, while in SMM, the entire RAM contents (extension over 4GB included).

4. We implement a QEMU-based [1] prototype, which enables us to show the usefulness and correctness of our solution as well as perform a performance evaluation.

2. RELATED WORK

Live memory acquisition is an interesting and challenging computer forensics topic, which has largely gained the attention of the research and industry community. In the following, we provide a brief overview of the state-of-the-art, pointing out its main characteristics and limitations.

2.1 Hardware-based Approaches

In principle, every PC hardware bus can be leveraged to gain access to the host physical memory via direct memory access (DMA). For instance, solutions relying on PCI [2], PCMCIA and FireWire [12] buses have been proposed in literature. Such techniques have the advantages of not causing any change in the state of a running OS and being unaffected by most of attacks-hiding techniques. Unfortunately, PCI devices require a prior installation on the system and this greatly reduces their usability. FireWire-based solutions address this issue by allowing analysts to hot-plug them in the target system; thus they can be carried in the toolkit of an incident response team to be installed just after an incident occurred.

Both techniques have limitations. First, they cannot access the processor state (i.e., registers). Second, a knowledgeable attacker, or an advanced malware, can perform a scan of the PCI bus and detect the presence of such ad-hoc devices and consequently decide to stop any malicious activity and wipe every fingerprinting left behind. Lastly, as shown by Rutkowska in [16], DMA devices can be tricked to provide split views of memory contents, thus making the output of such devices unreliable.

2.2 Software-based Approaches

Software-based approaches vary greatly both in complexity and reliability. Such solutions often rely on the OS internals of the host whose memory must be collected. As an example, the simplest way to dump the memory via software is to rely on special virtual devices, if present, like `/dev/mem` on Linux or `\Device\PhysicalMemory` on different Windows systems. Such devices, in fact, allow user space programs to read the whole physical memory of the running system. This possibility leads to trivial memory collection operations, for example through simple UNIX utilities such as `dd` and `netcat`. The main drawback of all these solutions is that they need to be loaded into memory in order to run, thus modifying the original state of the target machine. This causes the captured data to be inconsistent and not reliable. An alternative software solution, even if not always viable, is to crash the system that needs to be analyzed. Indeed, Windows automatically trigger a dump of the physical memory on the hard disk whenever a crash occurs.

Other software memory collection solutions can be used when dealing with virtualized environments. Indeed, virtualization opened up many new possibilities for forensic analyses. The execution of the virtualized system, commonly referred to as the *"guest"* operating system, can be completely frozen for an arbitrary amount of time, allowing for easy atomic collection operations.

One of the most advanced solutions in such a case has been proposed in HyperSleuth [11], where the authors describe a framework that leverages standard hardware support for virtualization to gather memory contents without interrupting the target services. The only drawbacks of this approach are that some small changes in the memory of the target are induced by the installation procedure and that a powerful attacker in the same network of the target could interfere with the packets containing the memory dump.

As previously mentioned, in the last years some authors proposed solutions which exploit the characteristics of System Management Mode. Among them, the closest to our approach is that presented in [21], where a mechanism for RAM collection leveraging SMM is presented. Such a mechanism however has been mainly devised for malware detection and thus does not address some critical issues required by digital investigations. More precisely, these includes the following problems: the integrity of the copy obtained, its adherence to the original content and the possibility of dumping the Physical Address Extension (if present). Furthermore, it requires the installation of a dedicated PCI network card. Our approach, as we will see, also requires a network card to operate, but it leverages the one already installed on the system and not a custom piece of hardware installed *ad-hoc* on the target.

In a subsequent paper [23], an extension of Hypercheck to comply with digital investigations has been proposed. However, no proof of concept has been provided and most of the limitations mentioned above have not been addressed.

3. SYSTEM MANAGEMENT MODE

SMM is a special mode of operation of Intel CPUs, introduced in the i386 processors, designed to handle system-wide functionalities, such as power management and hardware control.

The processors enters SMM in response to a System Management Interrupt (SMI), which has a higher priority compared with other interrupts, and it is signaled through the SMI# pin on the processor or through the APIC bus. When SMM is invoked, the CPU saves the current state of the processor, switches the System Management RAM (SMRAM) address space and begins to execute the code present in it (i.e., the SMI handler).

SMM code is not intended for general purpose applications, but it is limited to system firmware only. Indeed, the main benefit provided by SMM is the execution of the code in an isolated processor environment that operates transparently to the operating system. SMM is defined to be a real-mode environment with 32-bits data access when operand- and address-size override prefixes[1] are used. Otherwise, operand's and address's size is restricted to 16-bits.

The only way to exit from the SMM operation mode is by means of the `rsm` instruction that is available only in the SMM. The `rsm` instruction takes care of restoring the saved state of the processor, and returns the control to the interrupted program. When the processor is in SMM all hardware interrupts, but software-invoked interrupts and exceptions, are disabled.

During boot time, it is a duty of the BIOS firmware to intialize SMRAM, copy the SMI handlers to it, and, as a form of protection, lock SMRAM to disallow any further writing accesses to this area. In fact, SMM is the mode of operation with the greatest level of privilege, informally named ring -2, and has to be as safe as possible from malicious users. The solution presented in this paper works properly only if the SMRAM has been locked at boot time.

3.1 Threat Model

Before describing the technique we devised, we must delineate the threat model in which we expect SMMDumper to be used. Our model assumes the availability of an hardware method to trigger an SMI and trasfer control to SMMDumper, such as the one briefly depicted in Section 1. At the end of the section, however, we discuss what are the consequences of being unable to adopt an hardware solution.

The machine, which memory we must acquire, has allegedly been compromised and these conditions may hold:

- The attacker (or malware) has root access to the compromised system.
- The attacker has compromised other machines in the same network of the target.
- The attacker can perform network attacks (e.g., intercepting and modifying packets).

On the other hand, SMMDumper is not able to deal with the following situations:

- The attacker has access to the smartcard containing the private key used to sign collected memory.
- The attacker exploits vulnerabilities that allow *write* access to SMM memory of the target system [25].
- The attacker has physical access to the target system and is able to power it down or mount a DMA attack that wipes the memory before it is collected [24].

[1]Intel and AT&T syntaxes, respectively provide `A32`, `O32` and `addr32`, `data32` as explict address- (`0x67`) and operand-size (`0x66`) override prefixes.

We are well-aware that an attacker could write a malware able to thwart the execution of the memory dump, whether we use our fallback software-based solution to trigger an SMI. Specifically, the malware must have administration privileges in order to launch the attack, tamper our SMI triggering solution and subsequently prevent the launch of the dump of the system memory. To achieve its goal, the malware modifies I/O APIC Redirection Table by setting the delivery mode of the `IRQ1` to `Fixed` and the `Interrupt Vector` field to the malicious interrupt vector in the `IDT`. By doing so, the malware disables the ability of an user to trigger an SMI by pressing a specific keystroke that launches the memory dump. With respect to others techniques to trigger an SMI [22], the technique that we adopt, presented in Section 4.1, is more challenging to be silently disabled by a malware. This is due to the fact that it requires the modification of an offset of the well-known address of the I/O APIC Redirection Table. This behavior can be considered "suspicious" at least and may ring an alarm bell for common anti-virus software. This, of course, is far from being an optimal solution: as we already stated, the best solution would be a dedicated hardware method. Furthermore, to the best of our knowledge, every technique proposed so far that leverages SMM and needs to trigger an SMI on an allegedly compromised system, suffers from the same problem [22].

4. SMM-BASED MEMORY DUMP

This section describes in detail the design and implementation of SMMDumper, the SMM-based infrastructure we have devised to perform a consistent and unforgeable dump of the volatile memory of a running operating system.

As briefly outlined in Section 1, SMMDumper can be logically divided in two components: a triggering module and a memory collector module. The former component is responsible for invoking SMIs and thus entering SMM. Ideally, this component should be implemented in hardware to provide strong guarantees and resiliency against malware threats. Instead, we have opted for a software-based implementation to allow our solutions to be used on commodity hardware. The memory collector module represents the main component of SMMDumper. It is in charge of reading the physical memory of the target host and transmit it over the network. It is a BIOS extension loaded in SMRAM at boot time and unauthorized modifications of its content are prevented by having the BIOS locking write access to that specific region.

A global overview of SMMDumper architecture can be observed in Figure 1. Intuitively, a forensic analyst invokes SMMDumper by initiating a predefined keystroke sequence (1). This sequence is immediately intercepted by the triggering module, which switches the system CPU to SMM. The memory collector module (2) starts subsequently, initiating the host physical memory dump over the network (3). As we will see shortly, before the acquisition process actually starts, SMMDumper waits for detecting the presence of a commodity cryptographic device, which must be plugged into the system *after* entering SMM. This device is responsible for creating on-chip digital signatures and to provide strong integrity guarantees of the transmitted data (4).

Running code at system management mode privilege opens a number of challenges that need to be properly addressed to achieve the goals mentioned at the beginning of this section. In particular, we must (I) trigger system management interrupts to switch to SMM, (II) be able to access *all* the physical

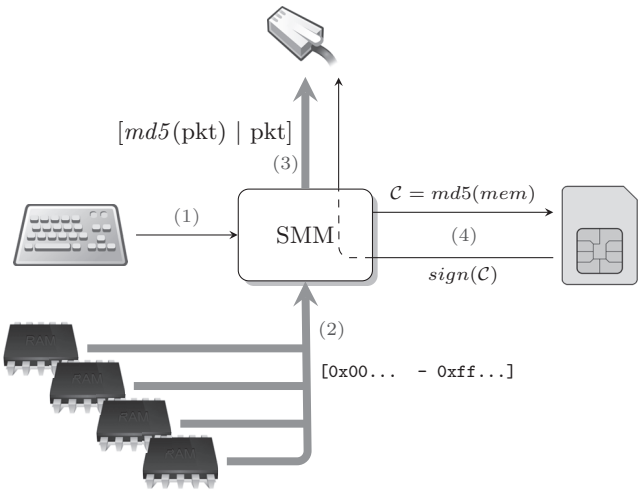


Figure 1: Overview of the system

memory of the target system, even when more than 4GB of physical memory is installed on 32-bit hosts, (III) guarantee the integrity of the collected data on the host as well as while in transit to a generic—but trusted—device. Meeting such challenges clearly guarantees the atomicity (I and II), reliability (III), and availability (III) forensic requirements illustrated in Section 1.

4.1 System Management Interrupts

Intel CPUs enter SMM by invoking a System Management Interrupt (SMI). SMIs can be triggered through either an external SMM interrupt pin (`SMI#`) or the Advanced Programmable Interrupt Controller (APIC). Even though only one SMI pin is physically hard-wired to the CPU, different events—generally specified by the I/O Controller Hub (ICH)—can trigger SMIs. Modern chipsets, such as the Intel ICH10 [5], have approximately 40 different ways to trigger an SMI, such as power management, USB, Total Cost Of Ownership (TCO), writing to the Advanced Power Management Control port register, periodic timer expiration and SMBus events. In addition, some motherboards are equipped with dedicated hardware that can be legitimately exploited to raise SMIs. For example, an SMM interrupt switch installed on the motherboard to allow users to suspend the system when turned on (power-save mode). Nonetheless, triggering an SMI requires a proper software register configuration.

Switching to SMM to start a whole-system memory dump requires raising an SMI whenever a specific keystroke sequence is detected. To this end, our approach builds on [3] to implement a fully-functional SMM-based keylogger. In particular, everything revolves around the Intel Advanced Programmable Interrupt Controller (APIC), which overlooks the communication between the CPU and external devices. The APIC is divided into I/O and Local APIC. They are located on the chipset and integrated onto the CPU, respectively, and communicate over a dedicated APIC bus. The I/O APIC receives external interrupt events from the system hardware and its associated I/O devices and, depending on the configuration of its Redirection Table, routes them to the Local APIC as interrupt messages. The Local APIC delivers the interrupts received from the I/O APIC to the CPU

```
1  start:
2    data32
3    movl  $0xdeadbeef, %eax
4    data32
5    addr32
6    movl  (%eax), %ebx
7    ;; Do something with %ebx
8    data32
9    addl  $0x4, %eax
10   data32
11   cmpl  %eax, $0x1000
12   jl    start
```

Figure 2: Accessing physical memory from SMM

it belongs to (after consulting the Local Vector Table, which specifies how interrupts are delivered to the CPU and their priorities). Finally, the Interrupt Descriptor Table (IDT) is indexed with the vector number by the CPU to select the proper Interrupt Service Routine (ISR) handler to invoke.

Overall, the Redirection Table plays a crucial role in the above-sketched process, as it specifies the interrupt vector and delivery mode of each interrupt pin. In particular, the delivery mode is fundamental to accomplish the goal of triggering an SMI when an arbitrary, custom and predefined, keystroke sequence is observed. To this end, we set the delivery mode of the `IRQ1`[2] to SMI and the vector information to 0s to properly forward that IRQ line to our SMI handler.

Our SMM ISR handler extracts the keyboard scancode from the keyboard controller buffer, reading from the I/O port `0x60`. Any scancode mismatching the predefined keystroke sequence is properly re-injected to the keyboard controller buffer by writing the keyboard controller command `0xd2` to the I/O port `0x64` [18].

Modifying the I/O APIC Redirection Table to deliver an SMI when the `IRQ1` IRQ line is asserted requires to forward the interrupt to the CPU. This is achieved by sending Interprocessor Interrupts (IPIs) from software by properly configuring the Local APIC Interrupt Command Register (`ICR`). To this end, we set the `ICR Destination Field` to `self` and the `Delivery Mode` to `fixed`. Writing to the least-significant doubleword of the `ICR` causes an IPI message to be sent out and the interrupt to effectively be delivered to the CPU as soon as the `rsm` instruction is executed.

4.2 Accessing Physical Memory

As noted elsewhere, SMM is similar to real mode. Therefore, the size of operands and addresses of the instructions executed in SMM are limited to 16 bits, restricting the addressable memory to 1MB. However, override prefixes are generally used to access up to 4GB of the addressable memory space [8], as briefly sketched in Figure 2. In particular, the snippet of code iteratively reads 4KB of memory starting from the address `0xdeadbeef`. It is worth noting that running code in SMM disables paging, allowing for a direct access to *physical*—rather than virtual—memory. While this has the clear benefit of allowing a straightforward memory access without worrying about virtual-to-physical address translations (and viceversa), it has drawbacks too. According to the Intel specifications [8], SMM can only access up to 4GB of physical memory, even on CPUs that support Physical Address Extension (PAE) or long mode with IA-32e. To overcome such a limitation, one may think about enabling paging in SMM to populate a custom Page Table to map physical-to-virtual pages and access them through virtual addresses. Unfortunately, enabling paging requires to switch the CPU to protected mode (the de-facto default mode of operation of Intel-like CPUs nowadays). Within SMM, this can only be achieved by executing the `rsm` assembly instruction, which causes an exit from SMM too [7]. We defer the solution SMMDumper adopts to Section 4.4.

[2]We intercept PS/2 and USB keyboard when its state is set as legacy mode.

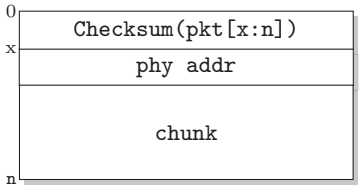


Figure 3: Packet format

4.3 Data Integrity and Transmission

Direct physical memory access alone does not meet all the forensics requirements sketched in Section 1. Ideally, SMMDumper could just read memory one byte at a time and send it off over the network. However, this would hardly represent a reliable solution. Running SMM code is also similar to running in hypervisor mode: there is no operating system service we can rely on and it is like being executed directly on the bare metal. Therefore, transmission errors may just occur and there are no in-place mechanisms to address such issues (e.g., data loss or integrity corruptions).

To overcome such limitations and meet the reliability forensic requirement outlined in Section 1, we have designed a simple, yet effective, communication protocol. Intuitively, SMMDumper divides the physical memory into chunks of fixed size (1KB in our current implementation). Each chunk is then embedded in a packet structured as shown in Figure 3. The base address of the memory chunk contained in the packet (i.e., `phy addr`) is metadata and represents a unique label that is used by the receiver to correctly handle out-of-order or missing chunks. Checksum over the whole packet payload (metadata and the physical memory chunk), instead, offers the opportunity to detect integrity violations.

4.3.1 Data Signing

Clearly, the simple checksum-based scheme outlined above protects only against transient network transmission errors (similar to what a TCP segment checksum does), but it easily fails against an attacker that purposely modifies data on-the-fly and recomputes the checksum to reflect such changes. To address this threat, SMMDumper computes an incremental checksum $\mathcal{C}$ of the whole physical memory, as individual packets are sent off[3]. Subsequently, when the transfer of the whole physical memory is completed, SMMDumper signs $\mathcal{C}$

[3]While details of the actual network transmission and retransmission due to loss or incorrect data are described next, let us just assume here that all the packets have been correctly transmitted to the receiver and that the whole system physical memory has thus been dumped.

and sends the resulting ciphertext blob to the receiver, which verifies the signature and compares $\mathcal{C}$ against a freshly computed checksum over all the received packets. A valid signature guarantees the integrity of the received signed checksum and matching checksums guarantee the integrity of all the received packets. To be able to incrementally compute $\mathcal{C}$, an appropriate algorithm needs to be chosen. As we will see in Section 5, SMMDumper uses the `MD5` cryptographic hash function.

The question of where to store the private key used for signing $\mathcal{C}$ still remains. Any attempt to embed the key into SMMDumper may be threatened by sophisticated attacks aimed at reading its code or data. To overcome such threats, we have decided to offload all the cryptographic operations (key management and signing) to an external hot-pluggable smart card device $\mathcal{D}$. In particular, as soon as SMMDumper starts executing (that is, as soon as the system enters into SMM), it waits for $\mathcal{D}$ to be plugged into the system to carry out the tasks outlined above. Not only this procedure guarantees the integrity of the collected memory, but it also provides validity as the signing key can only be accessed by whoever has access to $\mathcal{D}$ and can start the live forensic memory dump.

It is worth noting that hot-pluggable memory collection devices may induce tangible side-effects in the target memory as a result of their installation or initialization in the system, as pointed out in Section 2.1. However, SMMDumper requires the analyst to plug the smart card device once the system has entered SMM, where no operating system service is in execution and any potential side-effect is under the control of SMMDumper. In addition, different smart cards allow for the use of different signing identities, which can be easily produced and distributed to forensic analysts. By using a personal smart card, the forensic analyst implicitly certifies that a given live memory acquisition is associated to that specific, personal, smart card.

Assuming that errors do not generally occur, the schema just outlined is both effective and efficient as it generally requires only one encryption operation (signing) per a whole memory dump. In fact, signatures of individual packets would negatively impact on the overall overhead. On the other hand, if an individual packet is corrupted, the whole dump must be taken again. Although the errors-are-seldom-events assumption seems reasonable, SMMDumper can be easily extended to suite the analysts needs. For instance, signed checksums could be sent out for every 500MB of data, trading-off packet retransmissions and overall overhead.

4.3.2 Network Transmission

Once packets are ready, SMMDumper needs to transfer them from the target machine to an external trusted host or device, for future analyses. We have opted for the former solution. Conversely, the latter would, for instance, require to store packets on an USB storage plugged on the target machine and this raises two main concerns. First, interacting with USB devices from SMM is everything but simple and the code needed to interact with the USB controller would likely be bloated and prone to errors. Second, the destination USB device needs to be physically plugged on a port on the target system while, leveraging a network connection, the collected data can be sent to a remote host, tipically on the same local network but, *potentially*, even elsewhere on the Internet.

SMMDumper implements a basic network driver that is able to communicate through I/O operations with the Network Interface Card of the target machine. As noted elsewhere, operating in SMM does not allow to rely on any operating system-provided service, such as networking. Therefore, SMMDumper is also equipped with the code responsible to forge UDP packets sketched in Figure 3. Once again, the choice of the UDP protocol rather than the more reliable TCP is driven by the willingness of keeping the code as simple as possible and ease the burden of a fully-functional (and complex) implementation. As described next, this is not a limitation *per-se* as we do have enough metadata to recover from arbitrary transmission errors.

One could argue that transferring data over the network is less secure than using a removable device as the attack surface increases. Indeed, an attacker that compromises a machine in the same network of the target could use different techniques (e.g., ARP spoofing [20]) to intercept or prevent the reception of the packets sent by SMMDumper. However, it is impossible for an attacker to arbitrarily modify the content of a packet without being detected as the overall checksum sent at the end of the transfer is signed with a private key that, in our threat model, is inaccessible to the attacker. An attacker could still perform a Denial of Service attack, by dropping or damaging packets, but we argue that it is easy to identify this kind of attacks and its source inside a local area network and, consequentially, to exclude it from the network and then request a retransmission of the blocked packets through the protocol explained in the next paragraph.

4.3.3 Retransmission of Lost Data

As soon as SMMDumper finishes sending the memory to the remote host, it switches from send mode to listen mode, where it accepts requests to retransmit certain chunks of memory. The remote hosts uses metadata contained in the packets to check if everything was transmitted correctly, identifying in the process missing or corrupted chunks. Every lost packet is then asked back to the SMMDumper that recreates the packet and tries again to transfer it. An attacker, of course, could try to mangle with this mechanism, for example by modifying sent requests or forging fake ones. These are not really problematic attacks as the remote host knows which packets it has requested and can simply discard fake ones.

4.4 Accessing more than 4GB of Memory

There may be some situations in which SMMDumper is required to access more than 4GB of physical memory: if the CPU on the target machine supports Physical Address Extension (PAE), Page Size Extension (PSE-36) or IA-32e mode (namely 64bit support). PSE-36 is very similar to PAE, it just changes some internal structure of the page tables. Thus, in this paper we address only PAE among these two alternatives, as we believe that modifications needed by SMMDumper to handle PSE-36 would be trivial. Unfortunately, handling IA-32e mode is not straightforward and it is part of our ongoing research effort.

4.4.1 Handling PAE on IA-32

Physical Address Extension is a paging mechanism that is supported by an extension of physical addresses from 32 bits to `MAXPHYADDR` bits, where `MAXPHYADDR` is 36 bits on IA-

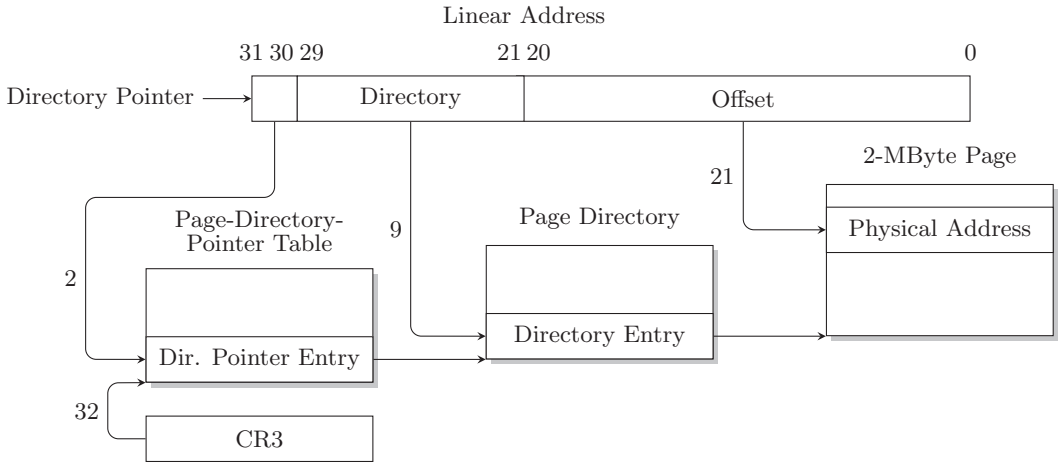


Figure 4: Address translation with PAE (2MB pages)

32 CPUs. Since our approach deals with IA-32 CPUs, from now on we will consider `MAXPHYADDR` to be limited to 36 bits. Theorically, this allows to use up to 64GB of memory. Unfortunately, as we said before, when we are in System Management Mode, we are limited to 32 bit addressing and, thus, we cannot use 36 bit addresses and registers. Furthermore, we cannot enable paging and switch to virtual addressing as this is also forbidden when in SMM. Some authors [25, 4] state that, surprisingly, it is instead possible to switch from SMM to protected mode, or even IA-32e. However, we did not find any specification that explicitly allows this switch and our experiments confirmed that it is not possible indeed. To the best of our knowledge, some CPUs may allow it but as we aim to reach a good degree of compatibility we do not rely anyhow on this possibility.

The rationale behind our solution is simple: once we finish dumping the first 4GB of physical memory, with the method explained in the previous Sections, we change the paging structure used by the target system so that we can control the mapping between virtual and physical memory, both from SMM and protected mode. Then, we inject some code in the mapped pages and we modify `EIP` field in the State Save Map so that it points to the code that we inject. Once everything is set up, we issue a `rsm` instruction and go back to protected mode. During the switch, `EIP` is restored from the State Save Map and thus the execution of the system resumes from the custom code we injected *before* the switch.

Setting-up Paging.

To better understand how we setup page tables before returning in protected mode, we reported the address translation mechanism with PAE enabled in Figure 4 as explained in [7]. For the sake of simplicity, we only analyze (and use) paging with 2MB pages. When PAE is enabled on a IA-32 CPU, the size of virtual addresses remains 32 bit while the size of physical addresses is extended to 36 bit. When translating a Virtual Address (VA) into a physical one, the MMU uses the structures depicted in Figure 4. The first two bits of the virtual address points to an entry of the Page Directory Pointer Table (whose physical address is stored in the register `CR3`). The MMU then checks this entry (PDPTE) to see if it is actually mapped (present bit) and if the required access is allowed; then, it follows the physical address contained in the entry, that points to the second structure: the Page Directory. Bits 21:29 of VA determine the Page Directory Entry (PDE) the MMU must use. The MMU performs the same check on the PDE and then it uses the base address contained in the PDE and bits 0:20 of VA to calculate the physical address corresponding to VA.

As can be observed, the whole address translation mechanism depends on the `CR3` register. When in SMM, however, we cannot modify the `CR3` register value stored in the State Saved Map. However, even if PAE is enabled, this register is *not* extended to 36 bits so we can access the memory pointed by the register, thus modifying the paging structure at our likings, even if we are restricted to 32-bit addressing. As can be seen in Figure 4, the `CR3` register points to the first Page Directory Pointer Table Entry (PDPTE). Part of the bits are used to control the Page Directory mapped by this PDPTE (e.g., if it is present, if caching is enabled) while the remaining part contains its physical address. Every PDPTE is 64 bit long, thus, to set it up correctly, we have to access it in two "rounds", using `CR3` for the lower 32 bits and `CR3+4` for 32 higher bits. SMMDumper edits the first PDPTE so that it points to a Page Directory located at physical address `0x0`. This implies that now the first Page Directory Entry (PDE) is located at `0x0`. SMMDumper configures this entry to be present, writable, accessible from every control privilege level and large (2MB instead of 4KB). The page base address of the first PDE is then set to `0x0`. This may seem wrong at a first glance, since we already use `0x0` as base address of the Page Directory. On the contrary, this is not only correct but also very convenient. Indeed, it means that we will be able to use *virtual* address `0x0` to access a 2MB physical page whose first 4KB correspond exactly to the Page Directory itself. This is possible because we edited the first PDPTE and the first PDE of the system, thus the MMU, when translating virtual address `0x0`, will walk these two entries, as we explained before, and will translate the virtual address into the physical address `0x0`. Furthermore, being able to access the Page Directory once switched to protected mode, by using virtual range `[0x00000000-0x00001000]`, will allow us to directly edit paging structures (i.e., map

new physical pages into virtual ones) without having to reference physical addresses.

Returning to Protected Mode.

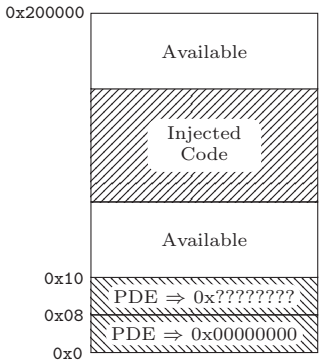


Figure 5: Layout of physical/virtual page `0x0`

Before returning the execution control to the system, further setup is required. First of all, we must insert the code that will be executed in protected mode. Obviously, it must be carefully tailored to avoid overlapping the memory areas we reserved for PDEs. This code is stored in the SMRAM along with the code needed to perform the dump of the first 4GB and then copied into a section of the first mapped page. This page has been mapped as explained in the previous paragraph, and its layout can be observed in Figure 5. As we can see, the first 16 bytes are reserved for PDEs: the first PDE maps physical page 0 (i.e., itself) on virtual page 0, thus allowing access to paging structure and to injected code even when SMMDumper will go back to protected mode. The PDE at address `0x00000008`, on the other hand, is prepared but still unused: SMMDumper will modify it to map physical memory pages that still need to be sent out (i.e., above 4GB). The injected code, depicted in Figure 6, performs the following tasks:

```
1   va       = 0x00200000
2   p_pde    = 0x00000008
3   phy_addr = 0x100000000 /* 36-bit */
4   while phy_addr < MAX_MEMORY :
5     /* Setup PDE */
6     p_pde->page_base_addr = phy_addr
7     p_pde->p  = 1 /* Present */
8     p_pre->us = 1 /* User/Super */
9     /* Now 0x00200000 points to phy_addr */
10    offset = 0
11    while offset < PAGE_SIZE:
12      packet =  str(phy_addr+offset)
13      packet += va[offset:offset+CHUNK_SIZE]
14      packet += MD5(packet[0:len(packet)])
15      /* Send pkt */
16      /* Update overall checksum */
17      offset += CHUNK_SIZE
18    phy_addr += PAGE_SIZE
```

Figure 6: Dump of the upper physical memory

- modify PDE at virtual address `0x00000008`, so that it will map a physical 2MB page that needs to be dumped. Since this PDE is the second one contained in the Page Directory under our control, this physical page will be mapped at virtual address `0x00200000` (lines 6 - 8);
- read a chunk of memory from *virtual* page starting at `0x00200000`, create a packet with checksum and send it over the network (lines 12 - 15). Alternatively, if we want to avoid code duplication, it is possible to just store memory chunks at an address that is readable from SMM (e.g., any area tagged as *Available* in Figure 5) and trigger a `SMI`. The code executed in SMM will then take care of signing and sending operations just like it did for the first 4GB;
- loop through the inner loop (lines 11-17) until a full 2MB page has been dumped, then repeat the procedure (lines 4-18) until the whole memory higher than 4GB has been dumped.

Before finally returning back to protected mode we have to make sure that (I) the execution flow will not go out of our code and that (II) our code has enough privileges to operate without triggering faults. To avoid (I) we clear the Interrupt Flag in the `EFLAGS` register stored in the Save System Map so that interrupts will not cause a transfer of the execution flow to the system interrupt handlers. Non-Maskable Interrupts (NMI), can still happen even if `EFLAGS.IF` is 0, so we disable them by interacting *directly* with the APIC. Disabling NMI is not a problem as the code that we will execute in protected mode does not rely on them anyhow. Problem (II) is mainly due to the fact that, from SMM, we cannot modify the CPL that will be set upon a resume. Fortunately, the only operations for which our code will need some custom privileges are I/O operations (both for interacting with the NIC or to trigger a SMI and get control back to code in SMM). This privileges can be granted by altering the Input Output Privilege Level (IOPL) bits in `EFLAGS` just like we do for `IF`.

4.5 Portability

The approach adopted by SMMDumper is *completely* OS-independent and can be easily applied as a patch to already existing BIOS or installed in new ones. The most restricting requisites of SMMDumper are mainly hardware:

1. The presence on the target system of a port (USB, serial) to interact with the smartcard hardware.
2. A network interface card to send out packets.

Developing drivers for many different network interface cards may be painful, as every piece of hardware has its specifications and peculiarities. However, most BIOS already include primitives to interact with on-board NICs (e.g., PXE [6] functionality required to interact with the network). BIOS manufacturers willing to support SMMDumper can easily modify it to include additional drivers of well-known on-board NICs.

5. EXPERIMENTAL EVALUATION

To verify the soundness of the proposed approach, we implemented a prototype based on coreboot [13], so that it can be easily installed on many production systems as a BIOS update. The collector module of SMMDumper has been entirely coded in Assembly and it counts slightly less than 500 LoC, of which $\approx$ 47% is for the `MD5` implementation. To perform tests, we employ QEMU 1.0.1 [1], equipped with a single Intel 3GHz processor, 6GB of RAM and a Realtek RTL8139 100Mbps network card.

To interact with the NIC, we implemented a small driver that is able to run in SMM to send and receive UDP packets. The code required to interact with this piece of hardware is quite simple: to transmit a packet, the driver writes the physical address and the size of the packet to the appropriate control registers of the device. Then, it polls the status register of the device until the transmission is completed. We use polling instead of interrupts because interrupts are disabled as soon as we switch to SMM. Packets reception, that is needed for the retransmission protocol, is implemented in the same way. It is worth noting that we allocate the packet transmission buffer in a region of memory outside the SMRAM because devices cannot access SMRAM due to hardware restrictions [17]. Before using this region, we backup it into the SMRAM and restore it later, in order to send it.

As a checksum algorithm to grant both packet and overall memory integrity we use `MD5`. As we briefly outlined in Section 4.3.1, our choice of `MD5` is convenient, as it allows us to *incrementally* calculate the overall checksum of the memory as we read it to create packets. To univocally sign the overall checksum, SMMDumper leverages an external hardware device that is able to read a private key from a smart card and use it to encrypt data. To perform experiments, we created a simulated smart card reader that can be hotplugged into our development environment by means of a Serial Port (RS232). According to our threat model, since the communication channel is established once SMMDumper is already executing, no attacks can interfere with this channel.

To evaluate the performance of SMMDumper and the average time needed to transfer the acquired volatile data over the local network, we performed a number of transmission tests (all the results are averages with negligible standard deviation). With a chunk size of 1024B of memory, each transmitted UDP packet carries a total payload size of 1024(`chunk`) + 16(`MD5`) + 8(`phy addr`) bytes. Using that value as our reference payload size, we measured the time needed to transfer 6GB of memory to be $\approx 820''$, $\approx 10\%$ of which is overhead due to the calculation of the `MD5` checksum over a single packet. The overall overhead caused by the metadata inserted in every packet is 144MB. It is worth noting that choosing a chunk size bigger than 1024B would introduce error-prone implementation intricacies in exchange for a relatively small gain. By maximizing the chunk size up to the MTU, indeed, we would incur in cross-page readings that, especially when dealing with PAE, may be tricky to deal with. Assuming a chunk size of $(MTU - TCP\ header - 16 - 8) = 1436$B, the time needed to dump 6GB and the overhead of metadata would respectively drop to $\approx 814''$ and ≈ 103MB. This roughly corresponds to an interesting downgrade of 29% of the overhead caused by metadata. However, when calculated on the whole traffic, the overall improvement is *only* of 0.66%. For this reason, our current implementation relies on a 1024B chunk size and trade a negligible performance boost for a more linear algorithm.

Finally, we must verify if SMMDumper is able to guarantee the properties that we illustrated in Section 1. Firstly, to evaluate if *atomicity* is satisfied, we instrumented our development environment in order to take a snapshot of the system memory before starting to execute our SMM ISR handler. Then, we compared such a dump with the one produced by SMMDumper. Results showed that our technique allows to gather an accurate and consistent memory dump: as we expected, no changes occur on the host after we trigger an SMI by pressing a specific sequence of keystrokes. However, we are aware that some changes may occur meanwhile SMMDumper reads memory mapped I/O regions which are reserved for devices. These changes however do not affect the *consistency* of our dump. Indeed, they cannot invalidate the overall checksum since it is incrementally calculated, i.e., we just read these areas once, both for sending them out and to calculate the hash. However, changes may happen between the time the user press the keystroke and the time at which the actual chunk is sent out over the network. Moreover, since we mapped SMRAM `[0x000a0000-0x000affff]` in a section of memory that overlaps the area used by video RAM `[0x000a0000-0x000bffff]`, the memory readings in that address range result in a dump of the SMI ISR handler and the SMRAM State Save Map, instead of the original content of the memory area. We assert that these changes do not violate *consistency* and *atomicity* properties as the information they contain are strictly related to physical devices and are not relevant for the analysis.

Secondly, we verified if the requirement of *reliability* is satisfied. In this part of the evaluation, we performed a man-in-the-middle attack between the target machine running SMMDumper and the receiver host [14]. The attack allowed us to accurately tamper with the packets, modifying the payload and re-calculating the `MD5` checksum to hide evidences of our modifications. The receiving host, at the end of the procedure, detected our evil modifications leveraging the overall encrypted checksum that SMMDumper sent at the end of the dump.

In summary, our experiments show that SMMDumper is able to provide an *atomic* and *reliable* snapshot of the target system, in a timely fashion.

5.1 Future Work

The actual SMMDumper implementation does not address Multi-Processor (MP) systems. Nevertheless, SMMDumper can be easily extended for handling it. In MP systems, SMI is propagated to all processors. As described in [15], processors enter SMM at slightly different times, because SMI could be serviced in between of CPU instructions. Therefore, SMM ISR handler waits for all processors to enter SMM and then, using a semaphore, only one processor executes the memory dump while the others wait it to complete.

Our threat model does not implicitly include the presence of malicious hardware-assisted hypervisors on the target system. However, when we do not have to dump more than 4GB of RAM, our solution is resilient to this kind of attacks. On the other hand, as we explained in Section 4.4, the fallback solution to dump higher memory must exit SMM with an `rsm` instruction. An hardware-assisted hypervisor could intercept the `rsm` [8] and get hold of the execution control before SMMDumper, thus disabling the acquisition of memory higher than 4GB.

Furthermore, we are currently working on a solution to tackle the problem of dumping even more than 64GB of RAM (i.e., on CPUs that support IA-32e mode). Preliminary results show that it is indeed possible to dump more than this amount of RAM. Furthermore, to the best of our knowledge, we believe that our future solution will be resilient to malicious hardware-based hypervisor attacks.

6. CONCLUSION

In this paper, we presented an SMM-based volatile memory acquisition technique that overcomes many of the limitations affecting state-of-the-art solutions. In particular, we have shown how SMMDumper is able to atomically perform a live memory acquisition, while guaranteeing the on-system and in-transit integrity of the acquired information.

While the firmware-based implementation of our proof-of-concept may be undermined by sophisticated kernel-level malware, the design of SMMDumper remains sound, arguing that the introduction of a naive and inexpensive hardware modification by vendors, such as an interrupt line directly connected to the processor SMI pin, would make SMMDumper completely bulletproof and resilient to any form of attacks.

Our experimental evaluation shows that SMMDumper is effective and efficient, allowing for its real-world deployment in digital forensic analyses and incident responses.

7. REFERENCES

[1] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[2] B. D. Carrier and J. Grand. A hardware-based memory acquisition procedure for digital investigations. *Digital Investigation*, 1(1):50–60, 2004.

[3] S. Embleton, S. Sparks, and C. Zou. SMM rootkits: a new breed of OS independent malware. In *Proceedings of the 4th International Conference on Security and Privacy in Communication Netowrks*, page 11. ACM, 2008.

[4] Intel. Intel Software Network. `http://software.intel.com/en-us/forums/showthread.php?t=63946`.

[5] Intel. *Intel I/O Controller Hub 10 (ICH10) Family*, 2008.

[6] Intel Corporation. Preboot Execution Environment (PXE) Specification, 1999.

[7] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3A*, May 2012.

[8] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3B*, May 2012.

[9] F. Z. Kun Sun, Jiang Wang and A. Stavrou. SecureSwitch: BIOS-Assisted Isolation and Switch between Trusted and Untrusted Commodity OSes. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, California, USA, February 2012.

[10] E. Libster and J. Kornblum. A Proposal for an Integrated Memory Acquisition Mechanism. *ACM SIGOPS Operating Systems Review*, 42(3):14–20, 2008.

[11] L. Martignoni, A. Fattori, R. Paleari, and L. Cavallaro. Live and Trustworthy Forensic Analysis of Commodity Production Systems. In *Proceedings of the 13th International Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2010.

[12] A. Martin. FireWire memory dump of a Windows XP computer: a forensic approach. `http://www.friendsglobal.com/papers/FireWire%20Memory%20Dump%20of%20Windows%20XP.pdf`.

[13] R. Minnich. coreboot. `http://www.coreboot.org/`.

[14] A. Ornaghi and M. Valleri. Man in the middle attacks demos. *Blackhat [Online Document]*, 2003.

[15] Phoenix. BIOS Undercover: Writing A Software SMI Handler, 2008. `http://blogs.phoenix.com/phoenix_technologies_bios/2008/12/bios-undercover-writing-a-software-smi-handler.html`.

[16] J. Rutkowska. Beyond the CPU: Defeating hardware based RAM acquisition tools, 2007. `http://invisiblethings.org/papers/cheating-hardware-memory-acquisition-updated.ppt`.

[17] T. Schluessler and P. Rajagopal. OS Independent Run-Time System Integrity Services. *Research Paper, IT Innovation and Research, Intel Corporation*, 2005.

[18] S. Sokolov. 8042 keyboard controller. `http://stanislavs.org/helppc/8042.html`.

[19] T. Vidas. *Acquisition and Forensic Analysis of Volatile Data Stores*. PhD thesis, University of Nebraska at Omaha, 2006.

[20] R. Wagner. Address resolution protocol spoofing and man-in-the-middle attacks. *The SANS Institute*, 2001.

[21] J. Wang, A. Stavrou, and A. Ghosh. Hypercheck: A hardware-assisted integrity monitor. In *Proceedings of the 13th International Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2010.

[22] J. Wang, K. Sun, and A. Stavrou. An Analysis of System Management Mode (SMM)-based Integrity Checking Systems and Evasion Attacks. Technical report, George Mason University, 2011.

[23] J. Wang, F. Zhang, K. Sun, and A. Stavrou. Firmware-assisted Memory Acquisition and Analysis Tools for Digital Forensics. In *Proceedings of the 6th International Workshop on Systematic Approaches to Digital Forensic Engineering (SADFE)*, Oakland, California, USA, May 2011.

[24] R. Wojtczuk. Subverting the Xen hypervisor. *Black Hat USA*, 2008, 2008.

[25] R. Wojtczuk and J. Rutkowska. Attacking SMM memory via Intel CPU cache poisoning. *Invisible Things Lab*, 2009.

Tapas: Design, Implementation, and Usability Evaluation of a Password Manager

Daniel McCarney David Barrera Jeremy Clark
Sonia Chiasson Paul C. van Oorschot
Carleton University, Ottawa, ON, Canada
{dmccarney,dbarrera}@ccsl.carleton.ca
{clark,chiasson,paulv}@scs.carleton.ca

ABSTRACT

Passwords continue to prevail on the web as the primary method for user authentication despite their well-known security and usability drawbacks. Password managers offer some improvement without requiring server-side changes. In this paper, we evaluate the security of *dual-possession authentication*, an authentication approach offering encrypted storage of passwords and theft-resistance without the use of a master password. We further introduce Tapas, a concrete implementation of dual-possession authentication leveraging a desktop computer and a smartphone. Tapas requires no server-side changes to websites, no master password, and protects all the stored passwords in the event either the primary or secondary device (*e.g.,* computer or phone) is stolen. To evaluate the viability of Tapas as an alternative to traditional password managers, we perform a 30 participant user study comparing Tapas to two configurations of Firefox's built-in password manager. We found users significantly preferred Tapas. We then improve Tapas by incorporating feedback from this study, and reevaluate it with an additional 10 participants.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*authentication*; H.1.2 [**Models and Principles**]: User/Machine Systems—*human factors*

Keywords

Password managers, usable security, smartphones

1. INTRODUCTION

A large number of research contributions have been made toward increasing the security and usability of password-based authentication [5]. Many of these attempts require account providers to change how they handle authentication by augmenting or outright replacing passwords; *e.g.,* one-time passwords, dual-factor, single-sign on, biometrics, graphical passwords, *etc.* Recently, researchers have argued that despite the wide-held sentiment from the security and usability communities that passwords need to be replaced, the incumbency, familiarity, and low cost of traditional passwords continues to hamper widespread adoption of an alternative, as well as a lack of consensus on what exactly the alternative should provide [12].

We are interested in practical solutions combining easy deployability with security and usability. For this reason, we presently exclude from interest proposals requiring server-side changes. Previous research under this constraint focuses on storing and retrieving passwords for users (*e.g.,* password managers), strengthening password quality (*e.g.,* randomly-chosen, cryptographically processed, or site specific), and encoding alternative authentication mechanisms into passwords (*e.g.,* graphical or object-based passwords). These three classes of solutions tend to address orthogonal issues and can be complementary. We focus on the first, not necessarily excluding the others.

Password managers are designed to relieve password fatigue and reduce log-in time. They can also indirectly facilitate better password quality and a reduction in password reuse. A naive password manager simply stores the passwords, while security-conscious managers lock the stored passwords under a master password. Password managers may also integrate other techniques to strengthen or encode passwords, including those mentioned above.

Password managers have certain drawbacks. To use a password manager, existing accounts must be migrated into the manager and potentially replicated across multiple devices. In the event an adversary gains access to the manager's storage, a naive password manager offers no protection making it a high value target. With a master password, the manager provides at best a level of protection dependent on the strength of the master password against an offline attack. This is assuming the theft does not occur when the manager has unlocked the passwords for the duration of a session, in which case the protection offered is greatly reduced. Password managers that maintain unprotected passwords during use do not always clearly indicate to the user the current state (locked or unlocked) of the system.

In this paper, we present a type of password manager that combines usability advantages of the naive password manager with protected storage. Passwords are protected

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

against offline attacks with a strong encryption key which the user need not remember and decryption requires the control of two independent devices. Operation of this type of manager requires no master password, only control of both devices. If any one of these two devices is stolen, the adversary cannot recover the passwords in practice.

We consider a specific instantiation of this type of manager, Tapas,[1] and present its design, implementation, and analysis. Tapas is a smartphone-assisted password manager for a computer that requires no server-side changes. It maintains security of the managed passwords by encrypting and storing the passwords on a smartphone, and keeping the decryption key inside the browser on the paired computer. Tapas is resistant to theft in the following sense: an adversary must steal both the smartphone and the user's computer to gain access to managed credentials. Tapas is designed to provide a simple mental model of "sending" the password from the phone to the login screen on a separate device, maintaining no cached master password and not storing any credentials on disk. Unlike a hashing-based solution, Tapas does not preclude memorization of passwords and login outside of the Tapas system.

We present the results of a 30 participant user study evaluating a Tapas prototype and comparing it to the built-in Firefox password manager both with and without the use of a master password. Our study found that in general users have little knowledge of the benefits password managers provide or the means by which they protect passwords. This leads to an underutilization of browser password managers and low enrollment in opt-in master password protection. Participants selected to use Tapas rated their enjoyment of the process higher than participants' ratings for the other managers. Further they were able to utilize Tapas successfully and without error to store credentials and log into websites, despite any perceived initial difficulties.

Our primary contributions are as follows:

1. We study the notion of *dual-possession authentication* which has received little attention in the literature. We develop a threat model for using it in conjunction with a password manager and find it offers a practical set of security and usability properties.

2. To allow concrete evaluation of this notion, we implement a dual-possession password manager (Tapas) using a Firefox extension on a primary device and an Android app on a secondary device. Although the idea of requiring two devices for password retrieval is simple, the implementation involves several subtle security and networking details. Tapas requires no server-side changes, no master password, and offers theft-resistance for the managed passwords.

3. We validated the feasibility of Tapas through an in-person user study with 30 participants comparing Tapas to two browser password managers. Users of Tapas were successful in using the system, even without prior knowledge of password managers. Using insights from the initial study we improve the Tapas design and then conduct a 10 participant follow-up study to evaluate it, finding it improves user's understanding of the system.

[1] Tap-based authentication using a smartphone

2. RELATED WORK

Recently, researchers have been encouraged to consider that the persistence of passwords is not incidental [12]: their advantages as a well-known, firmly entrenched incumbent (*e.g.,* widespread familiarity, marginal cost per user) outweigh the costs of implementing an alternative, and this is asserted to be unlikely to change in the near-term. Despite a wide-held sentiment from the security and usability communities that passwords must be replaced, there is little consensus on the actual harm incurred by password breaches [11] (passwords may not be the last line of defence), what fraction of breaches is attributable to each threat vector, and thus, what alternative schemes should prioritize.

Password Managers.

The category of password managers (client-side tools to assist password-based authentication) is broad and contains many different, generally complimentary, techniques. Examples of services they provide are password strengthening through iterated hashing [9, 16, 6], phishing protection through site-specific passwords [16, 19], and converting other types of authentication into passwords [14, 17, 2].

The other main service a password manager can provide is the storage and retrieval of passwords, which is the focus of this paper. Major browsers (*e.g.,* Internet Explorer, Firefox, Chrome and Safari) offer a built-in password wallet. These wallets store the passwords on the user's computer in either plaintext (often by default), or encrypted under a master password. The browser may also offer cloud storage protected by a typical user account (*e.g.,* password and recovery questions). Third-party applications like LastPass,[2] and 1Password,[3] focus on cross-browser, cross-platform support and cloud synchronization.

Wallets protected under a master password have two drawbacks. First, many implementations do not use encryption correctly—many mobile password wallets were demonstrated to be insecure[4]. A second drawback is that a user-chosen password may not resist an offline attack if the wallet is stolen. To address this issue, Bojinov *et al.* [3] propose the use of password decoys to force the adversary back to using online attacks.

Device-based Authentication.

Several papers have explored *device-based authentication*; we restrict our coverage primarily to those involving possession of a smartphone. With dual-factor authentication, a secondary token is required in addition to a password. One use of a smartphone in authentication is to generate such tokens (*e.g.,* Google Authenticator) or to receive them over SMS. Phoolproof [15] uses a smartphone as an authentication token to augment traditional password authenticaiton with the goal of preventing phishing through the use of public key cryptography and end-to-end TLS. Pico [18] uses a cluster of devices, including smartphones and other smart devices, in proximity of each other to allow authentication. All of these require server-side changes.

[2] `https://lastpass.com`

[3] `http://1password.com`

[4] A. Belenko and D. Sklyarov. "Secure password managers and military-grade encryption on smartphones: Oh, really?" *Blackhat Europe*, 2012.

Usability & Comparison Frameworks.

A number of papers have compared password managers through user studies. Gaw and Felten [8] surveyed users on password use and found few users employed a password manager instead of relying on memory alone. Chiasson *et al.* [7] examined two password managers, finding significant usability and security failings related to entry of the master password as well as inaccurate/incomplete mental models of the software. Bicakci *et al.* [1] examined the user interface of browser-based managers and the tendency of users to inadvertantly save private information on a public computer.

Karole *et al.* [13] performed a comparative user study between an online, a mobile, and a portable USB password manager. They found non-technical users preferred keeping their credentials on mobile phone based password managers, but had difficulty entering passwords of sufficient strength on the mobile device.

Bonneau *et al.* [5] propose a framework for evaluating authentication solutions based on usability, security and deployability properties. They rank 35 representative schemes (including 2 password managers: Firefox and LastPass). We evaluate Tapas using this framework in Section 8.

3. DUAL-POSSESSION AUTHENTICATION

Storing passwords, whether software-based or a post-it note with passwords written on it, is based on the principle of authentication by something you have: the contents of the password 'wallet.' The primary security vulnerability of an unprotected wallet is theft. This is traditionally addressed by adding a master password, something you know, for additional protection. However this protection is best considered a deterrent, as theft allows offline attacks on the master password. Given a user-chosen master password this may mean fewer than 20 bits of security [4].

By contrast, password management that requires simultaneous access to multiple paired devices offers a level of theft-resistance. Strictly speaking, this is not dual-*factor* authentication because the factors are of the same type: 'something you have.' For comparative reasons, we refer to it as dual-possession authentication. We assume that for most users, a large proportion of log-ins occur on a small number of devices. Dual-possession authentication is designed to improve the usability of the log-in process from these devices without negative impact on the rest.

Dual-possession authentication involves two applications, a Manager and a Wallet, on different devices and offers the three depicted protocols for managing the passwords: Pair (Protocol 1), Store (Protocol 2), and Retrieve (Protocol 3). These protocols are designed to achieve a relatively simple goal: by stealing the data of either the Manager or the Wallet, an adversary cannot determine the stored password for any given account with any greater success than attacking the account directly. This is achieved by encrypting each password with a key held by the Manager and storing the resulting ciphertext on the Wallet. By stealing the Manager, the adversary obtains the decryption key but not the ciphertexts to decrypt, and by stealing the Wallet, the adversary only has a set of ciphertexts resistant to offline attacks.[5] The effect of malware which remains resident on the Manager is discussed in Section 5.

To ensure these devices can run Store and Retrieve over a potentially hostile network, we require Pair to be performed on an authenticated and secret out-of-band (AS-OOB) channel [10]. The pairing is essentially an assignment of public keys that will be used by each device to authenticate the other during network communication. In Tapas we instantiate the AS-OOB channel by having the Manager display a QR code which is scanned by the Wallet. Once paired, the devices will establish a mutually-authenticated end-to-end secure channel (*e.g.,* TLS with a Diffie-Hellman key exchange[6]) before exchanging any encrypted passwords. This allows the devices to securely tunnel their communication through various network devices that may assist them in establishing a connection.

4. TAPAS

We instantiate the protocols and general notion of dual-possession authentication (Section 3) to construct Tapas. In Tapas, password management is handled across both the user's desktop PC and a paired smartphone. In this Section we describe the implementation details of Tapas, and explain how the 3 protocols of dual-possession authentication are enacted.

While we have chosen to implement the components of Tapas using Mozilla Firefox and the Google Android platform, the architecture is independent of these choices. We expect that an extension for Chrome, Safari, and other extensible browsers could be developed for users who do not use Firefox as their primary browser. Similarly, non-Android smartphone platforms could be used.

Firefox Extension. In Tapas, the Manager device is implemented as a Firefox browser extension on the users's desktop PC. It is written in JavaScript and XML User Interface Language (XUL), utilizing interfaces exposed by Firefox for use by extensions. It is multi-platform and requires no native code, allowing the extension to be installed on Windows, Linux or OSX.

Android Application. The Wallet device is implemented as an application for the Android smartphone platform. The Wallet is written using Java for devices running Android versions 2.3 and above. Based on platform distribution statistics[7] Tapas is compatible with over 81% of Android devices worldwide (as of Sept 4, 2012).

Rendezvous Server. In order to allow direct communication between two devices potentially located on separate networks, the Tapas architecture employs a Rendezvous Server to facilitate network address translation (NAT) traversal and hole punching. The Rendezvous Server is considered untrusted and external to the management of passwords; no unprotected data is transmitted through it.

In addition to negotiating network connections, the Rendezvous Server is responsible for federating communication with the Google Cloud to Device Messaging (C2DM) service. Google requires all applications utilizing C2DM to

[5]For two devices, this approach seems more straightforward than using distributed/threshold decryption with key shares.

[6]The RSA-based key exchange in TLS does not provide perfect forward secrecy, which is necessary for security as discussed in Section 5.

[7]Google platform versions distribution: `http://goo.gl/rQ2gv`

Protocol 1: Pairing Manager and Wallet

User action: Upon a user choosing to set-up a new Wallet, the following protocol is initiated by the Manager.

Communication channel: A one-way authenticated and secret out-of-band (AS-OOB) channel from the Manager to the Wallet.

1. The Manager generates an authentication key pair for itself $\langle pk_m, sk_m \rangle$ and sends its public key pk_m to the Wallet.
2. The Manager generates an authentication key pair for the Wallet $\langle pk_w, sk_w \rangle$ and sends the pair to the Wallet.
3. The Manager generates a secret key k for a symmetric key authenticated encryption scheme $\mathsf{Enc}_k()$.

Output: The Manager stores $\langle pk_m, pk_w, sk_m, k \rangle$ and erases sk_w. The Wallet stores $\langle pk_m, pk_w, sk_w \rangle$.

Protocol 2: Storing a Password

User action: Upon a user choosing to save a password p_i, the following protocol is initiated by the Manager.

Communication channel: A mutually-authenticated secure channel with perfect forward secrecy between the Manager and the Wallet. The participants, respectively, identify themselves with pk_m and pk_w.

1. The Manager takes user password p_i (entered by user) and site information s_i and computes $c_i = \mathsf{Enc}_k(p_i \| s_i)$.
2. The Manager sends $\langle c_i, s_i \rangle$ to the Wallet.
3. The Wallet prompts the user to create a tag t_i for referencing the site, using s_i to suggest a value for the tag.

Output: The Manager erases $\langle p_i, s_i, c_i \rangle$. The Wallet stores $\langle t_i, c_i \rangle$ and erases s_i.

Protocol 3: Retrieving a Password

User action: Upon a user choosing a password for retrieval, the following protocol is initiated by the Wallet.

Communication channel: A mutually-authenticated secure channel with perfect forward secrecy between the Manager and the Wallet. The participants, respectively, identify themselves with pk_m and pk_w.

1. The Wallet retrieves the c_i value associated with the tapped t_i, and sends c_i to the Manager.
2. The Manager decrypts and authenticates c_i to retrieve s_i and p_i.
3. The Manager checks that s_i matches the site information for the current site that the browser is visiting.
4. The Manager transfers the user password p_i to the site.

Output: The Manager erases $\langle p_i, s_i, c_i \rangle$.

pre-register with the service to obtain an API authentication token allowing access to the service. In order to avoid embedding a C2DM API token into the Manager extension we defer C2DM pushes to the Rendezvous Server, allowing the Manager to send a push message to a device through it. Tapas relies on C2DM strictly as a means of launching the Wallet application automatically without requiring a long-running listener service on the smartphone.

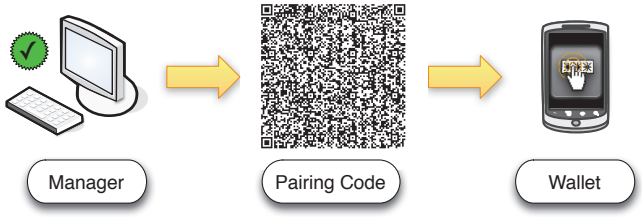


Figure 1: Setting up an out-of-band communication channel initiated (depicted by the checkmark) by the Manager, for pairing the devices.

4.1 Setup

To set up Tapas, the user installs the Firefox extension and the Android app using the standard software installation procedure for each respective platform. Once installed, the devices are paired using Protocol 1. The Manager computes the authentication key pairs and generates a self-signed TLS certificate for both public keys. It embeds networking information (IP address and port number), a fingerprint of its own certificate, and the Wallet's certificate and corresponding secret key into a QR code. The generated code is displayed on the computer screen, forming a unidirectional AS-OOB channel (Figure 1).

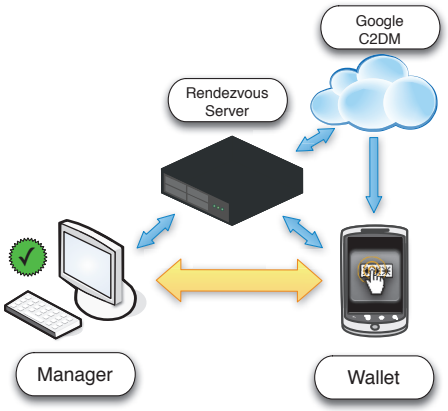


Figure 2: Setting up a two-way network communication channel initiated by the Manager, for password storage on the Wallet.

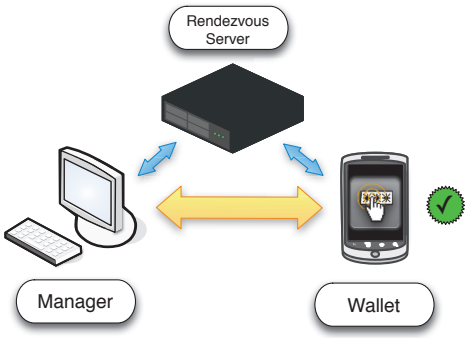


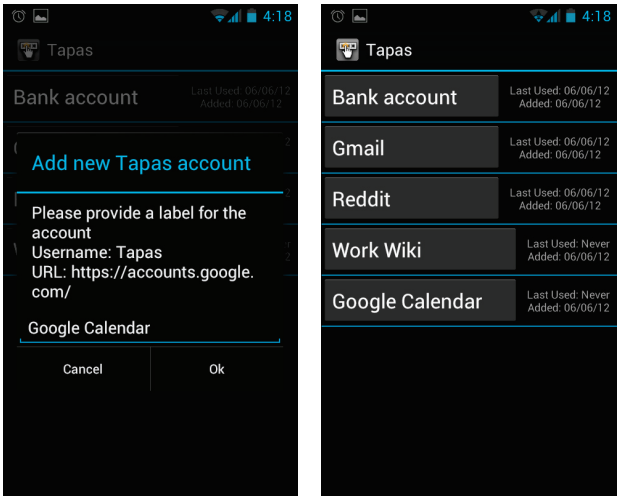
Figure 3: Setting up a two-way network communication channel initiated by the Wallet, for password retrieval from the Wallet.

The user now opens the Wallet app on the smartphone. The Android app defaults to displaying a "begin pairing" screen until the user successfully pairs the application with an instance of the Firefox browser extension. When the user presses the "pair" button on the Wallet app a QR code scan is initiated via the ZXing QR Code application.[8] ZXing utilizes vibration and auditory feedback to inform the user when a code has been successfully scanned in order to help make the process intuitive. After the Wallet app reads the QR code, the Wallet decodes from it the IP address and listening port of the Manager browser extension as well as the certificate material.

4.2 Account Import

When the Manager detects a username/password being submitted to a website, it temporarily saves the values as they are submitted and offers the user a chance to store the account credentials in the Wallet. This is done by presenting a non-obtrusive drop-down notification similar to the built-in Firefox password manager. If the user accepts the offer then the Manager contacts the Rendezvous Server to initiate a C2DM push to launch the Wallet application on

[8] `http://code.google.com/p/zxing/`



(a) Save account (b) Account list

Figure 4: Screenshots of the Tapas Wallet.

the paired smartphone (Figure 2). The smaller arrows represent communication used to launch the Wallet automatically (via C2DM and the Rendezvous Server) and to negotiate a direct network connection between the Manager and the Wallet (pictured as the larger arrow). Both the Manager and the Wallet rely on outgoing connections to the Rendezvous Server to negotiate direct communication through NAT, similar to traditional NAT hole punching techniques involving a third party.

At this point the Manager and Wallet follow Protocol 2 to securely transfer encrypted credentials. First the Manager encrypts the site information (URL, username, password) using AES in GCM mode with a symmetric encryption key known only to it. The encrypted ciphertext is then transmitted from the Manager to the Wallet over a mutually-authenticated TLS connection (using a Diffie-Hellman ciphersuite) where both certificates are pinned to the device certificates previously established during pairing.

When new account information is transmitted by the Manager to the Wallet the user is presented a chance to provide a meaningful label for the account (see Figure 4(a)). By default the label text is populated with the site URL; for privacy reasons the suggested label may be renamed. Each account in the Wallet has a large touch region displaying the user-chosen label for the account. Additionally, each account displays the date on which it was last used, and the date on which the account was added to the Wallet. Accounts are listed in order of most recent use (see Figure 4(b)).

4.3 Password Retrieval

When the user taps an account label in the Wallet on their smartphone, the stored credential associated with the account is transmitted to the paired Manager by Protocol 3. Figure 3 shows how the communication channel is set up. The Manager and the Wallet rely on communication with the Rendezvous Server (smaller arrows) to negotiate a direct network connection between one another (larger arrow). Assuming the user's browser is open to the correct website (*i.e.*, is viewing the URL associated with the tapped account) then the username and password field on the website are filled by

the Manager and submitted. The result of the login process is returned to the Wallet in order to display meaningful status messages to the user via the Wallet UI. All communication between the Manager and the Wallet is carried out over a mutually-authenticated TLS connection.

Tapas requires the user to signal their intent on both the browser and the smartphone before a login can occur. When a user transmits account credentials from the Wallet to the Manager, the latter decrypts the account information and verifies that the associated URL matches the currently open web page before filling the username and password. If a URL other than the one associated with the decrypted account details is open in the browser the Manager displays a message indicating that the correct URL must be opened before a login can occur (see Figure 5). This prevents accidental logins or a situation in which the user is away from their computer and accidentally triggers a login to a website by tapping their smartphone.

4.4 Limitations

The Tapas implementation is not without limitations. It relies on the availability of a network connection and the Rendezvous Server server to function. Given that the purpose of Tapas is authenticating with web resources the lack of an internet connection would likely preclude authentication regardless of Tapas. In order to use Tapas, both the paired devices must be present and usable. In the case of the smartphone Wallet this means the battery must be charged. The present implementation of Tapas allows pairing between only one computer and one smartphone, preventing use with multiple machines. Implementation of a full fledged secret sharing scheme could address the multiple device scenario.

5. SECURITY EVALUATION

We evaluate the security of Tapas relative to other types of password wallets, both with/without a master password. We assume the existence of an adversary with the ability to intercept, record, and modify any communication between the Manager and the Wallet except the one-time pairing process (Protocol 1) conducted over an AS-OOB channel (implemented in Tapas as a visible QR code). The pairing process allows the devices to establish public keys for authentication, enabling the devices to communicate confidently in the presence of an active adversary on standard communication channels (as in Protocols 2 and 3). In addition to granting access to the communication channel between the Manager and Wallet, we allow the adversary physical possession (theft) of either the Manager or the Wallet. Tapas offers no security against a loss of both.

Resistance to Theft.

If a device with an unprotected password wallet is lost, there is no inherent protection of the passwords stored in the wallet. The use of a master password offers some protection, however the adversary may still be able to conduct an offline attack that will recover all the passwords if the master password is not strong. On the other hand, a strong master password introduces usability issues related to memorability and accurate entry. In Tapas, theft-resistance is provided against offline attacks without the user having to remember any passwords.

Smartphones (which hold the Wallet in Tapas) are frequently lost and stolen. Passwords in the Wallet are encrypted in such a way as to be indistinguishable from randomness without the decryption key. This is a consequence of using the GCM mode of operation which provides indistinguishability under chosen plaintext attacks. The randomly generated 128 bit AES decryption key is held by the Manager and not contained on the smartphone, therefore the stored passwords are protected against even an offline attack. Further, aside from the user-chosen tag, all information about the sites that correspond to the stored passwords are also encrypted, providing privacy against individuals with passive access to the smartphone.

The Wallet also contains a wallet authentication key. Loss of this key to an adversary would allow the adversary to masquerade as the Wallet. The Wallet's only functionality is receiving and pushing encrypted passwords to and from the Manager. The ciphertext of each stored password is authenticated by the Manager's decryption key; a feature of GCM that prevents the decryption of any modified ciphertext. If a modified ciphertext caused the password portion to be submitted to a non-HTTPS site or one controlled by the adversary, the adversary could learn it. GCM does not allow the plaintext to be manipulated in structured ways, unlike other modes (*e.g.,* ECB or CBC). More generally, authenticated encryption ensures that the Manager cannot be a useful decryption oracle to the adversary.

The user's computer (which hosts the Manager in Tapas) may also be lost, stolen, or given away without the proper deletion of memory. In this case, the adversary recovers the symmetric AES encryption key k. k would allow the adversary to recover each password p_i given its ciphertext c_i, however the set of c_i are stored by the Wallet. Recall our assumption that the adversary can store all past communications observed over the secure channel in Protocols 2 and 3. Preventing such an adversary from learning the set of c_i is why it is essential that the encrypted passwords are communicated over an encrypted channel even though they are themselves already encrypted. Further, the adversary also learns the authentication key sk_m. If the design of the secure channel provided only authentication and encryption (using *e.g.,* the RSA-based ciphersuites in TLS), sk_m would be sufficient to derive the session key used in past executions of Protocol 2 and 3, allowing the adversary to recover the set of c_i. To thwart this line of attack, the secure channel in Protocols 2 and 3 have perfect forward secrecy (using a Diffie-Hellman key exchange in TLS) to ensure past session keys cannot be derived from a compromised sk_m.

Resistance to Malware.

Like other password managers, Tapas cannot protect stored passwords from persistent malware on the user's computer. The passwords must, at some point, be in plaintext for submission to the web service as per the current design of most web services (this is true if users memorize their passwords as well). With a traditional password manager, malware can immediately recover all the stored passwords as soon as the master password is entered. With Tapas, individual site passwords can only be recovered as they are used. If the malware is detected and removed, unused passwords will remain safe by repeating Protocol 1. Tapas also provides protection against specific forms of attack like hardware keystroke loggers and shoulder surfing.

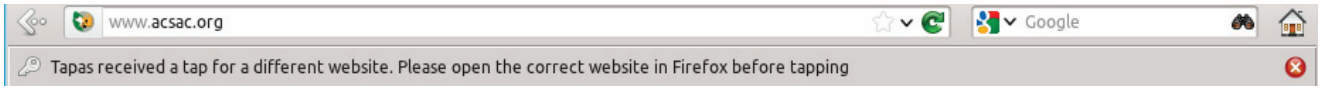


Figure 5: The notification the Tapas extension displays for mismatched user intent.

6. USABILITY EVALUATION

To evaluate the usability of Tapas, we conducted an in-lab user study with 30 participants. (Our later 10 participant follow-up study is presented in Section 7.2.) Our study design was approved by our university's Ethics Review Board.

6.1 Overview

We selected a between-subjects design where participants were randomly assigned to one of three conditions: Firefox with no master password (NMP), Firefox with a user-chosen master password (MP), and Tapas. Each participant was asked to complete a set of core tasks using the assigned password manager (see Section 6.5). We collected data through observation of the participants' interaction with the password manager as well as through questionnaires before and after each session. We did not mention to participants that Tapas was our own application so as to not bias participants.

We opted for an in-person study rather than a Mechanical Turk study for two reasons. First, Tapas requires the use of an Android phone and installation of an app not available in the market. In our study, we provided participants in the Tapas condition with an Android phone pre-loaded with the application. Second, conducting an in-person study allowed for direct observation of user behaviour when using the password managers.

6.2 Participant Demographics

We recruited a total of 30 participants (17 males, 13 females) through posters around the university campus and mailing lists. Most (age 18 to 42, $\bar{x} = 24.13$) were university students or staff. Participants had a wide range of backgrounds including accounting, psychology, theoretical physics, criminology, music, computer science and math.

Devices, operating systems and browsers. The majority of participants described themselves as Windows users (86%) and Google Chrome users (76%). A smaller number used MacOS and Linux regularly and one participant did not know how to tell what operating system he/she used. Chrome was the most popular browser, followed by Firefox which was used regularly by 50% of participants. Internet Explorer, Opera, and Safari were less popular. 28 participants owned a cell phone or smartphone. Smartphone OSs were approximately evenly split among Android, iOS and Blackberry.

Passwords and password managers. When asked to describe their use of passwords on the Internet, participants reported having between 3 and 40 ($\bar{x} = 11.53$) accounts that require passwords, and between 2 and 25 ($\bar{x} = 5.73$) unique passwords for those accounts, implying password reuse. 70% reported changing their passwords very rarely or never. 2 of 30 participants commented that the only time they change their passwords is if they are forgotten.

Participants in general had a poor understanding of the term *password manager*. Only 2 reported using a password manager, but several explained during their session that they do in fact use the browser's built-in password manager.

6.3 Study Setup

An Ubuntu Linux computer with Firefox pre-installed was used to perform the study. The Firefox history and settings were restored to defaults between sessions, so every participant saw the same "clean install" version of the browser.

Tapas requires that both a Firefox extension be installed on the desktop PC and an Android application be installed on the smartphone. We chose to avoid testing this common software installation process, and focused on the initial (post-install) setup and use. Thus, the Tapas Manager extension and Android Wallet app were installed, but not configured, before user sessions.

Three blog websites were created for the study (hereafter referred to as blog A, B and C). Each blog was designed to require an account (login) for posting comments.

6.4 Session

In-person sessions lasted 25 minutes on average and participants were paid $10. Participants were asked to read and sign an informed consent form which stated that their passwords would be logged, but not disclosed. Each participant was asked to read a short explanation of password managers in general, followed by a description of the specific password manager selected for their session. These text descriptions[9] were written with the objective of helping the user build an accurate mental model of the password manager rather than focusing on technical accuracy.

6.5 Tasks

During each session, participants were asked to perform the following tasks, after being given verbal instructions only at the start of each task (the examiner's involvement being minimal thereafter).

1. **Configure password manager:** If applicable (*i.e.,* in the MP and Tapas conditions), perform initial configuration. For MP, enable the master password protection in the Firefox settings and create a master password. For Tapas, scan the QR code displayed in the Tapas → Preferences pairing screen.

2. **Create and store accounts into the password manager:** Visit blogs A and B, find the register or create account section and select a username and password. When prompted, save the account into the password manager.

3. **Migrate an existing account into the password manager:** Participants were given a username and password and asked to pretend they already had an account on blog C. Proceed to log in to blog C and save the account into the password manager. Log out of blog C.

[9]Available at `http://pdox.ca/tapasscript`

4. **Log in to blogs:** After a distraction task,[10] participants were asked to log into and comment on the three blogs, in the following order. First log in to blog C. Next, close and reopen Firefox. Next, log in to blog A, followed by blog B. Closing and reopening Firefox was done to help users (particularly those in NMP and MP) identify when their passwords were accessible. Firefox, when configured with a master password, prompts the user for the master password on the first login after the browser is opened.

7. RESULTS OF FIRST USER STUDY

After completing the in-lab tasks, participants were given a post-test questionnaire designed to capture their comments and experience while interacting with the password manager. This section presents the questionnaire results and observations made by the examiner during the sessions.

Statistical tests. The Kruskal-Wallis non-parametric one way analysis of variance test was applied with a p-value of 0.05 considered significant to determine whether responses from participants in each condition were independent for a given question. If this test yielded a statistically significant p-value, one of the conditions was independent. To determine which condition(s) were independent, individual pairs were further analyzed with a non-parametric Mann-Whitney test to identify the specific conditions that differed.

7.1 Post-test Questionnaire

Perceived usability of password manager setup. Overall, there were no issues with the setup process in any of the three conditions. Participants in all conditions rated the ease of setup (on a 4-point Likert scale) as either easy or very easy with no obvious trend. Application of the Kruskal-Wallis test found no significant evidence that Tapas differed from the other managers in ease of setup.

Participants were asked if they thought they would be able to set up the password manager on their own. In all conditions participants responded positively. Participants in the MP condition noted that while the setup process was easy, finding the master password checkbox in the Firefox preferences was not straightforward, and that if the session information did not guide them to the right setting it would have been more difficult.

Comments from Tapas participants included the following: "This was a really easy step, I had never done it before but it was extremely simple", "It was pretty straightforward. It is easy to use", "The use of the QR code was a great tool to pair the devices. The set up was easy and quick".

Participants in the NMP condition (no setup required) were also allowed to enter comments regarding setup. A few voiced concerns about the simplicity of setup, stating that it was almost "too simple", and that you may actually end up accidentally saving passwords with the manager you didn't intend to save, a concern echoed in the literature [1]. Comments like these reinforce the Tapas design feature that requires signalled intent on both devices prior to saving account information or logging in using the password manager.

The QR code pairing method appears to be very intuitive and the audio and vibration feedback was verbally noted by some participants as useful. Tapas users mentioned feelings of accomplishment, as though they had achieved something complicated with little effort. On the other hand, the Firefox master password setup screen displays a password meter which no user was able to fill. Some users typed in two or three different passwords to try to increase the measure of the password strength bar.

Perceived usability of password saving. We asked participants to rate their agreement (on a 5 point Likert scale) with the following statement: "Saving a password was easy when I created a new account and migrated an existing account". Participants did not find Tapas any more difficult than the other two conditions, although some participants had to be reminded to complete the saving process on the phone after clicking the save button in the Tapas extension.

We also asked participants if they thought that using the password manager they were assigned made logging in easier than logging in without one. Based on verbal feedback from participants in the MP condition it appears some users perceive lower ease of use due to the master password being requested when the password manager is invoked for the first time. For the Tapas condition, one participant verbally noted that "logging in with Tapas would take longer since you would have to take out your phone and launch the app every time you log in".

User affectation. We asked users to rate how much they enjoyed using the password manager overall. Participants liked Tapas more than MP, and liked MP more than NMP. For this question, the Mann-Whitney test resulted in statistically significant difference between the Tapas condition and both the MP and NMP conditions ($p = 0.04891$ and $p = 0.006826$ respectively). For the NMP condition, 6 participants reported enjoying the password manager and one participant highly disliked it. In the MP condition, 5 participants enjoyed using the system, and 5 "somewhat enjoyed" it. Participants in the Tapas condition universally (10/10 participants) rated their enjoyment at the highest level of the Likert scale, demonstrating high user affectation in comparison to both the MP and NMP conditions.

General participant observations. In the free-form comment field at the end of the survey, several participants across all conditions expressed a desire to know where the passwords were stored. Some participants in the Tapas condition did not notice the pop-down message asking them to save their passwords. Considering this input, we modified Tapas as described in Section 7.2.

A second major theme of comments was related to losing access to the password manager. Several users stated that they probably would never use a password manager because if they lost access to it, they would lose access to all their accounts. While in reality users could still employ the password recovery mechanisms offered by individual websites, these comments highlight the importance of addressing a loss-of-access scenario. This motivates future work to enable an encrypted backup feature for Tapas.

7.2 Improving Tapas – Follow Up User Study

We revised the Tapas Firefox extension incorporating feedback from our 30 participant user study, specifically address-

[10]The distraction task had participants count down from 100 in decrements of 3 to help remove the recently created passwords from the participants' working memory.

ing issues with the poor visibility of the pop-down messages. For the message offering the user the chance to save an account (user ID, password) with Tapas the background was changed from gray to blue, and the label was changed from "Save with Tapas" to "Save to phone". The error condition pop-down messages were changed to have a red background. We revised the help text used in the Android Wallet application to clarify the goal of the pairing process.

To test the usability of the revised version of Tapas, we recruited 10 (6 female, 4 male) additional participants for the Tapas condition only. The study methodology was identical to the previous study, with the only change being the updated Tapas Firefox extension. For the most part, participants in the new study provided confirmation of the earlier ease of use and affectation findings. For brevity, we only present noteworthy results.

User attention. Observing participants during the second study confirmed that the blue message attracted participant's attention to the "save password" prompt. One participant remarked that the font size for the message was too small. Only one participant had to be reminded to look for the pop-down message after registering an account.

Mental model. The post-test survey attempted to capture the mental model participants had while using the password managers. All 10 participants in the second study answered correctly when asked "Where were your passwords stored?". In the first study, only 6 of 10 participants in the Tapas condition correctly mentioned the phone. While not statistically significant with 10 participants, we believe a larger sample would likely demonstrate that renaming the save button to "Save to phone" had a strong impact on the users' understanding of how the password manager works. The updated button label clearly explains where passwords are going when the button is clicked. In contrast, participants in the NMP and MP conditions answered this question correctly 50% of the time. Incorrect answers included some participants stating passwords were stored "in cyberspace", "on the website memory" and "no idea". We attributed this to Firefox's ambiguous "Remember password" button label.

7.3 Ecological Validity

Regarding demographics, most participants reported using Chrome as their primary browser, as well as not using a password manager. Thus, the lab study introduced these participants to both a new browser and a new password manager. This may have overloaded participants' memory, moving their attention away from the password manager or otherwise introduced a confounding effect.

The websites used were purpose-built blogs, and thus account integrity was not highly valued by users. Participant interaction with these sites, particularly in relation to password choice, may have been influenced by the lack of personal importance the blogs offered.

Some participants mentioned that the websites used in the study behaved strangely while logging in. Our sites were designed with minimal functionality requirements. Thus, when a user successfully logged in, the login form would be replaced with a message saying "successfully logged in", rather than returning to the password protected resource. Some participants failed to notice the change in login status, and were confused because they thought the login had failed.

8. COMPARISON SUMMARY

In this Section, we evaluate Tapas using the Usability-Deployability-Security (UDS) framework of Bonneau *et al.* [5]. Table 1 rates Tapas on the 25 benefits (properties) comprising the framework. For space, we cannot compare Tapas to all 35 authentication methods presented in the UDS paper. We focus on the incumbent (ordinary passwords) and the Firefox password manager from our user study.

Tapas addresses the usability issue of recalling an ever-growing number of passwords, without resorting to reuse. Relative to ordinary unmanaged passwords this benefit comes at the cost of interacting with a smartphone. In the event that access to the smartphone is lost, passwords need to be individually recovered using existing recovery mechanisms. Adding accessibility features to Tapas for disabled users is future work. When using a password manager, the stored passwords themselves can always be attacked directly. For this reason, password managers cannot improve on certain security properties of passwords. Tapas does provide phishing protection by ensuring stored passwords are only ever submitted to the exact site (determined by the site's URL and SSL/TLS certificate) they were registered with. Additionally the password cannot be observed externally when a user logs in with Tapas.

Surprisingly Firefox without a master password and Firefox with a master password rate equivalently using the UDS framework except for two properties: Memorywise-Effortless and Physically-Effortless. With a master password Firefox receives an empty circle rather than a filled circle due to the recall/entry of the master password. While a master password increases the security of stored passwords in the event of unauthorized access (*e.g.,* computer theft, in-person use, or through exposed backups) the UDS framework is not fine-grained enough to distinguish this security benefit. We opt to discuss just the master password version of the Firefox manager in Table 1.

Relative to Firefox MP, Tapas does not require a master password but does require interaction with a smartphone. On security, Tapas offers resilience to external observation. The framework does not distinguish that the stored ciphertexts in Tapas are resilient to an offline attack if the wallet is stolen, whereas in MP an offline attack on the master password coupled with access to the browser reveals all stored passwords at once. Similar to Firefox MP, Tapas receives an empty circle in the Physically-Effortless[11] and the Nothing-to-Carry[12] columns. Additionally, malware capable of recovering the Firefox master password can immediately learn all stored passwords, while malware on Tapas results in the gradual disclosure of passwords only as they are used.

For many cases, Tapas does not preclude composition with other mechanisms for improving security. For example, it can be used for per-site hash-based passwords, randomly generated passwords, or remembering the password portion for dual-factor authentication. Tapas could additionally generate a backup of the wallet's stored ciphertexts protected by a master password for recovery.

[11]We consider scrolling equivalent to pushing a button per the Physically-Effortless definition. Removing the phone from pocket is equivalent to removing a YubiKey or similar dongle.

[12]We consider the Tapas desktop component beyond the scope of Nothing-to-Carry, as Pico-siblings are likewise ignored.

Scheme	Memorywise-Effortless	Scalable-for-Users	Nothing-to-Carry	Physically-Effortless	Easy-to-Learn	Efficient-to-Use	Infrequent-Errors	Easy-Recovery-from-Loss	Accessible	Negligible-Cost-per-User	Server-Compatible	Browser-Compatible	Mature	Non-Proprietary	Resilient-to-Physical-Observation	Resilient-to-Targeted-Impersonation	Resilient-to-Throttled-Guessing	Resilient-to-Unthrottled-Guessing	Resilient-to-Internal-Observation	Resilient-to-Leaks-from-Other-Verifiers	Resilient-to-Phishing	Resilient-to-Theft	No-Trusted-Third-Party	Requiring-Explicit-Consent	Unlinkable
	Usability								Deployability						Security										
Tapas	●	●	○	○	●	○	●		○	●	●		●	●	○						●	●	●	●	●
Passwords			●		●	●	○	●	●	●	●	●	●	●	○							●	●	●	●
Firefox (MP)	○	●	○	○	●	●	●		●	●	●		●	●	○						●	●	●	●	●

Table 1: Evaluation of Tapas using the Usability-Deployability-Security framework for comparative evaluation of password alternatives [5]. The second and third rows are taken from [5] for reference comparison to Tapas.

9. CONCLUSION

We view the proliferation of "always keep me logged in" options and the interest in password managers and federated identity as evidence that the repetitive recall and typing of passwords is unpleasant for users. We designed Tapas to be compatible with password-based authentication, while relieving users of traditional password memory burdens. Tapas avoids the use of a master password— a setting which users found difficult to locate on existing password managers, does not offer strong protection against offline attacks, and may be inadvertently disclosed by users. Additionally with Tapas, users can walk away from their computers without exposing stored passwords that may be temporarily unlocked—every login requires an explicit action by the user. In the future, we intend on continuing the development of Tapas, as well as exploring other dual-possession schemes, specifically to facilitate logging in to accounts on mobile devices.

10. ACKNOWLEDGEMENTS

We thank Joseph Bonneau, Nitesh Saxena and the anonymous reviewers for useful feedback. This research is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC)—the second author through a Canada Graduate Scholarship; the third through a Postdoctoral Fellowship; the fourth and fifth through Discovery Grants and as Canada Research Chairs. We also acknowledge partial support from NSERC ISSNet.

11. REFERENCES

[1] K. Bicakci, N. B. Atalay, and H. E. Kiziloz. Johnny in internet café: user study and exploration of password autocomplete in web browsers. In *Digital Identity Management*, 2011.

[2] R. Biddle, S. Chiason, and P. C. van Oorschot. Graphical passwords: Learning from the first twelve years. *ACM Computing Surveys*, 44(4):1–41, 2012.

[3] H. Bojinov, E. Bursztein, X. Boyen, and D. Boneh. Kamouflage: Loss-resistant password management. In *ESORICS*, 2010.

[4] J. Bonneau. The science of guessing: analyzing an anonymized corpus of 70 million passwords. In *IEEE Symposium on Security and Privacy*, 2012.

[5] J. Bonneau, C. Herley, P. C. van Oorschot, and F. Stajano. The quest to replace passwords: a framework for comparative evaluation of web authentication schemes. In *IEEE Symposium on Security and Privacy*, 2012.

[6] X. Boyen. Halting password puzzles – hard-to-break encryption from human-memorable keys. In *USENIX Security*, 2007.

[7] S. Chiasson, P. C. van Oorschot, and R. Biddle. A usability study and critique of two password managers. In *USENIX Security*, 2006.

[8] S. Gaw and E. W. Felten. Password management strategies for online accounts. In *SOUPS*, 2006.

[9] J. A. Halderman, B. Waters, and E. W. Felten. A convenient method for securely managing passwords. In *WWW*, 2005.

[10] T. Halevi and N. Saxena. On pairing constrained wireless devices based on secrecy of auxiliary channels: the case of acoustic eavesdropping. In *CCS*, 2010.

[11] C. Herley. So long, and no thanks for the externalities: the rational rejection of security advice by users. In *NSPW*, 2009.

[12] C. Herley and P. C. van Oorschot. A research agenda acknowledging the persistence of passwords. *IEEE Security & Privacy*, 10(1):28–36, 2012.

[13] A. Karole, N. Saxena, and N. Christin. A comparative usability evaluation of traditional password managers. In *ICISC*, 2011.

[14] M. Mannan and P. van Oorschot. Digital objects as passwords. In *HotSec*, 2008.

[15] B. Parno, C. Kuo, and A. Perrig. Phoolproof phishing prevention. In *Financial Cryptography*, 2006.

[16] B. Ross, C. Jackson, N. Miyake, D. Boneh, and J. C. Mitchell. Stronger password authentication using browser extensions. In *USENIX Security*, 2005.

[17] N. Saxena and J. H. Watt. Authentication technologies for the blind or visually impaired. In *HotSec*, 2009.

[18] F. Stajano. Pico: no more passwords! In *Security Protocols*, 2011.

[19] K.-P. Yee and K. Sitaker. Passpet: convenient password management and phishing protection. In *SOUPS*, 2006.

On Automated Image Choice for Secure and Usable Graphical Passwords

Paul Dunphy
Culture Lab
School of Computing Science
Newcastle University
paul.dunphy@ncl.ac.uk

Patrick Olivier
Culture Lab
School of Computing Science
Newcastle University
p.l.olivier@ncl.ac.uk

ABSTRACT
The usability of graphical passwords based upon recognition of images is widely explored. However, it is likely that their observed high memorability is contingent on certain attributes of the image sets presented to users. Characterizing this relationship remains an open problem; for example, there is no systematic (and empirically verified) method to determine how similarity between the elements of an image set impacts the usability of the login challenge. Strategies to assemble suitable images are usually carried out by hand, which represents a significant barrier to uptake as the process has usability and security implications. In this paper, we explore the role of simple image processing techniques to provide automated assembly of usable login challenges in the context of recognition-based graphical passwords. We firstly carry out a user study to obtain a similarity ranked image set, and use the results to select an optimal per-pixel image similarity metric. Then we conduct a short-term image recall test using Amazon Mechanical Turk with 343 subjects where we manipulated the similarity present in image grids. In the most significant case, we found that our automated methods to choose decoy images could impact the login success rate by 40%, and the median login duration by 35 seconds.

Categories and Subject Descriptors
K.6.5 [**Management of Computing and Information Systems**]: Security and Protection: Authentication

General Terms
Experimentation, Security, Human Factors

Keywords
Usability, Security, User Authentication

1. INTRODUCTION
Users of alphanumeric passwords are widely known to choose credentials that are sufficiently predictable to undermine their principal security benefits [17]. This has led to the proposal of strategies of password selection designed to mitigate such predictability such as passphrases and mnemonics [19]. However, the success of these strategies, relies upon their adoption by conscientious users, and the strategies themselves may even contribute to user choice biases of their own. Graphical passwords [2][36] based upon the recognition of previously seen images have been proposed and evaluated in order to ascertain their suitability as a viable alternative to passwords and Personal Identification Numbers (PINs). Systems have been empirically evaluated with a password space ranging from 11 bits [13] to over 50 bits [40], and results highlight promising usability. The principal benefit of such systems is thought to be that users are more likely to remember assigned authentication credentials, thereby eliminating issues relating to predictable user choice. This genre of system provides a simple user interaction, as users must perform a visual search to identify previously assigned images (key images) amongst decoy images. The sequence of key images comprises the graphical password. Systems initialized using personal images [26,37,38] are particularly promising due to the ubiquity of large image collections, in addition due to observed usability benefits where users have had involvement to capture or create the images [28].

However, it appears likely that the usability benefits of this genre of graphical password are contingent upon subtle attributes of the image sets that are presented to users. For instance, it has been noted that visual and semantic similarity exhibited between images has the potential to disrupt the *picture superiority effect* [32] by causing errors in visual search [7] and rote learning [34]. For security and simplicity, it would be desirable that image sets are assembled randomly; however, the constraints that clearly exist when creating a usable login challenge imply that some degree of skill and effort is required to do this effectively, and in a manner that preserves security. Figure 1 illustrates different extremes of assembling decoy images for a particular key image. Zurko and Simon [43] remind us that user-centered security should be proportioned between end- users, developers and system administrators. Currently such a holistic consideration is lacking, as there is currently no systematic or empirically verified convention to reason over the similarity in an image set, and as a result this

Figure 1: Extremes of decoy image selection for the same key image: left) decoys are semantically different; centre) semantic and visual similarity to key image; right) decoys are semantically similar yet different from the key image.

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

process must be performed *by hand* on the basis of common sense judgments of image semantics [1]. Users themselves could be asked to tag for similarity, but this can present security threats if users attempt to circumvent the process to obtain an overly simplistic login task (e.g. Figure 1, right image). This absence of a systematic means to evaluate image similarity, combined with the potential impact of inappropriate levels of similarity on authentication error rates, constitutes a significant barrier to the real world deployment of graphical passwords. Indeed, the ability to spontaneously generate usable image sets likely could present security benefits in terms of guessabilty, phishing and observation attack, and becomes more pressing when considering likely deployment level phenomena such as password resets, where new image sets would need to be generated in response to user demand.

An as-yet unexplored approach to solve this problem is to harness image processing research from the field of *content-based image retrieval* (CBIR) [12]. One fundamental challenge is to determine whether two images are similar. In this field the underlying assumption is that images with similar visual characteristics are more likely to be semantically similar [39]. Our contribution is to explore the efficacy of a systematic method to identify instances of visual similarity between images, and explore the impact of its careful manipulation upon the usability and security of images presented in a graphical password login. We carry out two user studies: firstly we gain a human consensus on the visual similarity within an image set, and use those results to identify a mechanism to detect digital image similarity that can best represent our collected human consensus. Then, in a second study we use the most promising method to generate graphical password logins for a user study conducted on Amazon Mechanical Turk. The study captured participant performance in a short-term recall task where the visual similarity present in the login was manipulated to differing degrees. Finally, we consider the security implications of systematic filtering, identify attacks that can result, and propose countermeasures.

2. RELATED WORK

2.1 Graphical Passwords

Recognition-based *graphical passwords* [36][2] have received increasing research attention due to experimental results in cognitive psychology that suggest the existence of a *picture superiority effect* [32]: that images are retained in memory better than words or numbers. This has led to a number of instantiations of user authentication systems that aim to harness this effect. Passfaces [25] harnesses human ability to recognize faces. In a field study across three months, it was discovered that users of this system made one third of the errors made by those using traditional passwords [4]. Findings from the field suggest that asking users to remember a specific ordering of the images causes errors [1][18], that the rate at which users are introduced to new systems can impact retention [10], and that the memorability of images increases the more involvement the user has in their creation [28]. Tullis and Tedesco [38] performed a series of studies on a photograph-based system and found users to accurately recognize personal photographs amongst stock photographs, even when stock photographs were hand-picked to be semantically similar to their own images. The same authors also reported impressive user recall performance six years after the initial study[37].

Image processing is beginning to play a role in the usability and security of graphical passwords. Dirik et al. [6] proposed image processing techniques that could serve to predict user choice in the Passpoints [41] system. Focusing upon two particular images, they performed object segmentation, calculated probabilities for the objects users would likely focus their attention upon, and calculated the centroids of those objects to serve as candidate click points. By doing so they were able to guess 80% of user click points they collected in a user study. Salehi-Abari et al. [31] developed these ideas further and used corner detection to identify candidate click points, and optimized their guessing strategy according to techniques likely to be adopted by users in graphical password selection. They assembled a 34.7 bit attack dictionary and were able to guess (depending on the image) 48% to 54% of entire click point sequences. Dynahand [27] considered the problem of eliminating visual similarity from image sets where the image stimuli took the form of freehand doodles.

2.2 Similarity and Visual Search

There are a range of definitions of similarity. Medin [22] suggests that: "*Similarity between two things increases as a function of the number of features or properties they share and decreases as a function of mismatching or distinctive features*". Smith et al. [34] used the Farnsworth-Munsell 100 Hue test in a paired associate learning context to explore the impact of visual similarity upon rote learning. Here, colors with different controlled levels of similarity were assigned textual tags and participants were challenged to recall the tag when presented with a color. They found that when stimuli were similar, the performance scores for recall were worse and suggest that rote learning decreases as a function of the number of stimuli relevant for a particular textual tag. In the field of visual search, what we refer to as key images are referred to as *target* images, and decoys are referred to as *non-targets*. Duncan and Humphreys [7] explored the efficiency of visual search when using alphabetic characters and hand manipulating the choice of distracters to varying degrees of similarity. They concluded that there exist two types of similarity: *within-object conjunctions*: where the spatial arrangement of strokes is similar (e.g. L vs. T); *across-object conjunctions:* where the target can be formed by recombining strokes from different non-targets (e.g. find R amongst a set of P and Q characters). In addition, they identify interesting properties of visual search: that it takes longer to decide that a target is absent than to say it is present, and that target images can be camouflaged if placed next to similar non-targets.

2.3 Digital Image Processing

The most common way to compare digital images for similarity is to create and store signatures of an image that represent particular features e.g. color, and compute distances between these signatures. Choosing the most appropriate image signature is a context specific task. Color is the most widely used attribute in image retrieval and object recognition [20]. The SIMPLIicity system [39] attempts to systematically categorize the image type before choosing the most appropriate feature representation. Histogram-based methods are widely studied and are considered to be effective. Rubner et al. [29] propose the use of the *Earth Mover's Distance* (EMD) as a method for calculating a distance between two color histograms for purposes of determining their similarity. This reflects the minimal cost of transforming one distribution into another. Lv et al. [21] explore using a modified version of the EMD for similarity matching and propose the *average precision* metric. Pass and Zabih [24] suggest adding multiple dimensions to histograms to include characteristics other

than pure color, such as spatial information. Selection of the most appropriate color space can also be an important decision, as some are more perceptually linear than others and so lend themselves better to reasoning over image similarity [11].

3. IMAGE FILTERING AND GRAPHICAL PASSWORDS

We firstly define some terms: the *image set* comprises all the images available to the authentication system; the *login challenge* is a subset of the image set, which is comprised of both key images and decoy images and is presented to the user at login. There are a number of conventions regarding the presentation of the login challenge to users, however for simplicity we constrain our discussion to the mode where the login challenge is displayed across a sequence of grids, and where one key image is certain to appear in each grid. *Image filtering* is the process of reducing an image set into a login challenge through a process of choosing key images and their associated decoy images.

The absence of an accepted automated process to perform image filtering has likely, in part, motivated recent research pursuing the identification of an optimal image type for recognition-based graphical passwords [15] (see Table **1**). This optimal image type is intuitively one that minimizes the burden placed upon a person to undergo the process of image filtering by hand, and one that allows users to perform favorably in recall tests with the login challenges assembled using that process. The drive to satisfy both constraints has led to a focus upon particularly contrived image types that by design permit a narrow range of possible interpretations as to their content (e.g. clipart). The lack of explicit attention given to image filtering even in these contexts appears to assume that it is a one-off procedure, and that the resulting login challenge can be reused for each user of the system. However, likely realities of deployment might make the use of a finite image set unrealistic. For instance, inevitable password resets would mean that previously seen images must be discarded from the image set for a particular user. In addition, if the image set or login challenge is static between users, then attackers can build up knowledge regarding user behavior with those images e.g. user choice, can permit phishing, and spontaneous distribution of credentials e.g. password sharing, observation attack (due to the images providing a common frame of reference shared between users). This approach also takes little account of results that have suggested users have better memory retention for images they have created [28], nor context-specific defenses that result from strategic selection of image content [9].

There is a general lack of knowledge regarding the impact of strategies of image filtering upon usability and security. The assumption so far has been that images should all be semantically and visually different for purposes of usability. A different strategy is illustrated by Passfaces [25], a commercial system where the image stimuli are drawn from a database of normalized face photographs. A brief description offered regarding their image filtering procedure is the following: *"The grids of faces in Passfaces are grouped by sex and are selected to be equally distinctive so that Passfaces cannot be described by gender or obvious characteristics"*[25] pg. 5. This illustrates sensitivity to risks of large semantic differences in the login challenge and the ability for users to share the graphical passwords. Usability concerns must result, however one study of human memory involving 2500 images presented in pairs showed that participants

Table 1: Illustrative results from graphical password studies focused upon different image types.

System	Stimulus	Entropy	Success %
Komanduri [18]	Icons	35 bits	100%
Passfaces [4]	Faces	12.7 bits	85%
Tullis & Tedesco [38]	Photos	20 bits	100%
Dynahand [27]	Scribbles	9.5 bits	99.4%
Weinshall [40]	Clipart	47 bits	>90%
Déjà vu [5]	Fractals	16 bits	90%

could be accurate at remembering precise image details. Even where images were visually and semantically identical and exhibited only small differences in detail, e.g. orientation, user recognition rates were only marginally worse than when images exhibited semantic differences [3]. The assembly of a login challenge based upon distinct semantics is perceived to improve usability, but those assembled to incorporate similar semantics could be harnessed to improve security.

In either case, while the curation of a usable and secure login challenge remains a skill residing with those with the greatest experience of doing it, the propagation of such systems more generally is limited. The spontaneous use of everyday uncurated image collections (e.g. photographs) in this context is perceived to be particularly challenging, however, this image type is in some ways attractive, as sets of uncurated images are ubiquitous in personal collections and online. It is possible that if methods of automated image filtering based upon judicious analysis of image content are identified, this could reduce the imperative to identify an optimal image type.

3.1 Image Similarity in the Login Challenge

There is currently little convention to follow regarding where to apply systematic analysis of image similarity. Figure 2 outlines points in a typical recognition-based graphical password login challenge that could comprise the image filtering procedure. Analysis can occur on a *per-grid* and a *per-login* basis. On a per-grid basis, *intra grid key-decoy* similarity refers to the similarity between a key image and collocated decoy images. The most usable visual search is one where decoy images appear distinct from the key image [7]. High similarity in this dimension suggests that users might confuse the key image with a collocated decoy image, whereas low similarity suggests the key image would appear to be easier to identify amongst the decoy images. *Intra grid decoy similarity* refers to the difference between collocated decoy images. In isolation such consideration provides few

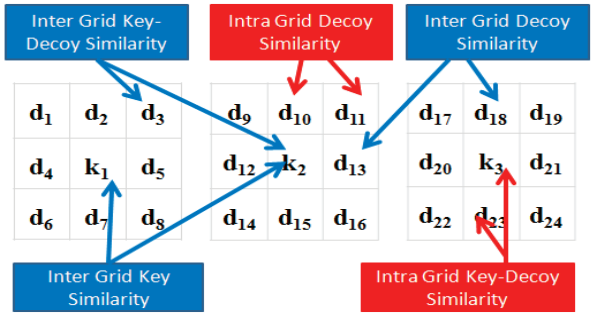


Figure 2: Points at which to consider image similarity across an example login challenge. Red indicates a per grid consideration, and blue indicates a per login consideration.

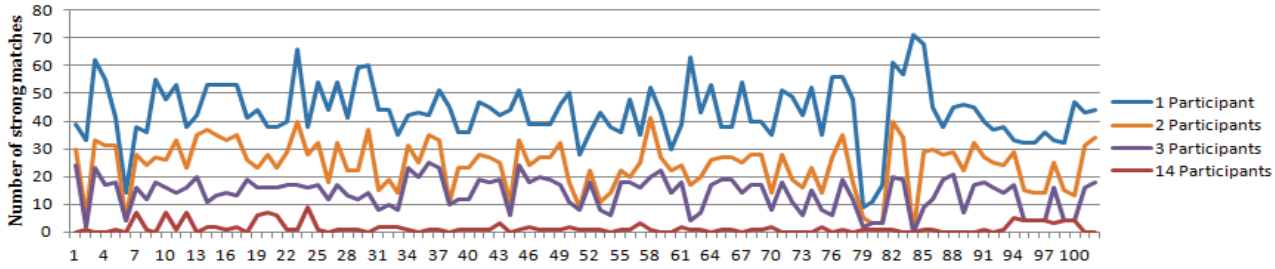


Figure 3: Visualisation of the number of strong matches identified per image in the first study. A strong match is determined by a threshold upon the number of participants that must have classed an image pair as being similar. Overall: 1 participant=4424 strong matches; 2=2462 strong matches; 3=1425 strong matches, 14=148 strong matches.

usability issues, however high intra grid decoy-decoy similarity and high intra grid key-decoy similarity indicates that a grid may overall appear visually similar, which could complicate usability, observation attack and description [9].

The per-grid consideration should be complemented with per-login analysis. *Inter grid key similarity* refers to the similarity between key images. If there is high similarity in this dimension there is a threat that an attacker might infer that pattern (e.g. all key images are of specific objects or contain particular colors). If this difference is too great, it is likely that images will be more difficult to remember for the user who must remember visually and semantically disconnected images. *Inter grid key-decoy similarity* refers to the similarity between a decoy image and non-collocated key images. This is important from a usability perspective, as a decoy image may appear to be similar to a non-collocated key image and entice users to select it erroneously. *Inter grid decoy* similarity considers the similarity of decoy images across a whole login challenge. High similarity in this regard, along with high *intra grid decoy similarity,* indicates that decoys across the whole login could appear visually similar, whereas high *inter grid decoy similarity* and low *intra grid decoy similarity* indicates that there exists similarity within the decoys in each grid, however each grid appears visually different.

4. USER STUDY 1 – HUMAN CONSENSUS OF IMAGE SIMILARITY

Perceptions of image similarity are subjective. However, in order to measure the success of a proposed image processing intervention, it is necessary to first obtain some ground truth notion of pairwise similarity that exists within a particular image set. To do this we carried out a user study to capture a human consensus of similarity within an image set to provide the basis for further study.

4.1 Procedure

We assembled a set of 101 digital photographs and recruited 20 participants (14 male, 6 female with ages μ=27, σ=6) who were staff and students in the research lab. Each participant was asked to organize the printed set of 101 images into piles on a tabletop according to a similarity ranking method proposed elsewhere [35]. This involved the participant being asked to organize the set of images into piles, with the only criteria being that those perceived to be similar should be placed in the same pile. No further advice is offered. The raw data per-participant were the image numbers present within each pile. Across all participants this was aggregated into a score n for each image pair (x,y), where $(x,y)=n$ means that image x and y appeared on the same pile n times, where $n \leq 20$, and high values of n indicate high agreement of similarity. The set of 101 images was intended to be representative of a typical photograph collection. The size of the image set was chosen to provide a manageable sorting task for participants. The images were printed onto high quality paper (100mm x 80mm) and the reverse of each was numbered. For descriptive purposes only we labeled the images according to the following informal categories: *People* (9): focus is a person or group of individuals; *Scene* (30): the focus is purely a landscape scene; *Object* (14): the focus is purely an object; *People/Scene* (47): the focus is upon people and scenery; *People/Object* (1): the focus is upon both people and an object. The image collection contained images taken to a wide range of photographic quality, and was sourced by aggregating a number of personal collections.

4.2 Results

Figure 3 gives an overview of the raw output for this study, which highlights the subjective nature of image similarity judgments even across a relatively small image set. For each image, the graph illustrates the number of other images considered to be a strong match for similarity. For a pairing to be considered a strong match, we applied a threshold to n that represented the minimum number of times images should have been placed on the same pile. As we increase the threshold, fewer images are classed as a strong match. The median number of piles participants sorted the images into was 21.5 (IQR = 12.25) with a minimum of 6 piles and a maximum of 32 piles. Images in the *people* and *scene* categories generally had the highest number of image matches (Median=45, IQR=13) and those in the Object category had the least (Median=33; IQR=23). No systematic investigation of the strategies used to group images was conducted, but in general it was apparent that these ranged from matching particular objects in the image, to matching the overall context, contrast level or principal colors. Although we only use 101 images in the results, the graph shows 102 (the original number) since one image was misplaced in the course of the study (#84).

4.2.1 Choosing an Automated Similarity Measure

A final phase of this study was to test a number of image processing methods to identify one that was most appropriate to detect the most severe instances of similarity as identified in the sorting task. The threshold of n≥14 (that is: in our first study, fourteen or more participants judged two images as similar) provided a basis for us to identify the most severe cases of similarity that any automated mechanism should detect. The field of *Content-based Image Retrieval* is fast moving; our approach was to test a number of candidate image signatures that would not require extensive expertise in image processing to understand and

implement. We reused the images from the first study and performed analysis with those images in the CIE(L*a*b*) color space [11] which is more perceptually linear than RGB or HSV. We implemented each of the following in OpenCV:

- **Statistical Moments**: treat each channel of a digital image as a probability distribution and calculate the first three statistical moments. To compare two image signatures we calculated the Euclidean distance between the statistical moments of each color channel and threshold the result.
- **Color Histogram**: the histogram bins contain the frequencies of particular pixel values. Firstly we initialize a histogram with 16x16x16x16 bins, which divides each 8 bit color channel into 16 bins. In the normalization phase, each bin is set to a value between zero and one representing its relative frequency with regard to the other bins. Then we remove any bins with less than 1% of the volume as this can be attributed to noise. To compare histograms we calculate the (EMD) [30] which treats the histograms as *piles* and provides the minimum cost of turning one pile into the other. The threshold for similarity was 0.9.
- **PerceptualDiff** [42]: this is not an image signature but is a suite of algorithms that contains a model of the human visual system. Its canonical task is to optimize the computer graphics task of global illumination, by determining whether two scenes are perceptually similar. We were interested to see if a more sophisticated approach held promise.

For each method we made a single pass of the digital images from the sorting study where each was resized to 384x286. We took each image in turn, calculated the corresponding image signatures and compared to every other image in the set, noting the images that were judged to be similar in each case. To calculate the success of these routines, we employed widely used metrics for information retrieval: *recall* and *precision*:

$$\text{Recall} = \frac{|\{\text{relevant images}\} \cap \{\text{retrieved images}\}|}{|\{\text{relevant images}\}|}$$

$$\text{Precision} = \frac{|\{\text{relevant images}\} \cap \{\text{retrieved images}\}|}{|\{\text{retrieved images}\}|}$$

We had knowledge of *relevant* images from the first user study in the form of the strong matches identified by participants for each image. *Retrieved* images are the set of images that the particular method judged to be similar. The metric of recall provides a measure of the fraction of relevant images that a particular method returned. The precision provides the fraction of the returned images that were relevant and is sensitive to false positives. Where thresholds had to be chosen to make a decision of similarity for a particular image processing intervention, they were selected to balance precision and recall. To incorporate spatial information into the calculations we also augmented the statistical moment and color histogram methods with a vertical or horizontal *region of interest* (ROI). This involved partitioning images with a vertical or horizontal line, calculating the image signature for both halves and using the mean of the two as the result for that image. Table 2 summarizes the filtering results obtained for each method. In addition to precision and recall we calculated the F_1 score, which is used to aggregate both precision and recall and represents a weighted average of the two.

Overall, the color histogram image signature applied to whole images provided the best recall at .58. The addition of spatial information to the image signature through the vertical ROI gave higher recall than the horizontal ROI but also introduced more false positives. The use of statistical moments was less effective than the color histogram in all configurations, as recall was .34, and this in fact dropped with the introduction of ROI, although ROI eliminated false positives. PerceptualDiff produced a lower recall than both color histogram and statistical moments. The use of PerceptualDiff was most effective at returning very strict matches where visually the objects and colors in the scene appeared similar. As might have been anticipated, the recall using the ROI was consistently lower than signatures based upon whole images, but ROI augmentation also yielded fewer false positives. This was reflected in a high score for precision.

Table 2: **Results from filtering procedure on an image set with 800 photographs, resized to 384x286.**

Method	Recall	Precision	F_1
Color Histogram	.58	.95	.4
Color Histogram & Vertical ROI	.48	.8	.3
Color Histogram & Horizontal ROI	.41	1	.3
Statistical Moments	.34	1	.3
Statistical Moments & Vertical ROI	.20	1	.2
Statistical Moments& Horizontal ROI	.07	1	.1
PerceptualDiff [42]	.24	.9	.2

Since the color histogram approach provided the best recall and the highest F_1 score, we chose to use this in our second study. The efficacy of the color histogram is likely because that representation captured the diversity of color without being restrictive spatially. This method did not provide a perfect recall score; however, we believed this score was difficult to better given the set of images in use.

5. USER STUDY 2 – RECALL TEST USING AUTOMATICALLY SELECTED IMAGES

The first user study suggested that the optimal image signature we tested was the color histogram, as it provided the closest predictor of the human similarity judgments we collected. Our remaining research question concerned whether systematic manipulation of thresholds chosen for this image signature could have a predictable impact upon the short-term recall of users in a typical graphical password login.

5.1 Procedure

We chose a between-subject study design where the independent variable was the similarity between a key image and its decoy images, and the dependant variables were user performance in terms of recall and login time. We developed a web-based system that would challenge the user to identify four key images across four grids of nine images in a 3x3 layout, with one key image certain to appear in each grid, providing theoretical entropy of 12.7 bits. We chose three experimental conditions where similarity between the key image and its decoy images was controlled by a threshold upon the EMD distance *d* between the color histograms of the images:

- **Similar**: where 1>d>=0
- **Middle**: where 4>d>= 3
- **Dissimilar**: where 6>d>=5

Studies in psychology [7] have observed how the difficulty of the visual search should decrease with decreasing similarity between

Figure 4: The key images chosen for the study, these were the same in all study conditions.

target and non-targets. We were hoping to recreate a similar trend. To generalize our results more effectively we firstly discarded the image set used in the first study and obtained a set of 1000 images used in other image processing research [39], and removed any portrait oriented images for display consistency (reducing to 800). The database is highly categorical, which provides a worst case scenario for this study. In advance, we also chose 8 key images that represented exemplars of particular categories in the image set (see Figure 4). These key images were persistent across conditions. For each key image and experimental condition we automatically chose eight decoy images according to the condition-specific similarity criteria. We also enforced distances between other images in the login to respect the image filtering concerns discussed in Section 3.1. Within a particular condition, once an image was selected as a decoy to be associated with a particular key image, it could not be selected to appear as a decoy image for a different key image within that condition. Also, a key image could not reappear as a decoy image. Figure 5 provides an example of decoy image selection for one particular key image across all three conditions.

We recruited participants from the crowdsourcing platform *Amazon Mechanical Turk*. Kittur et al. [16] provide hindsight from conducting user studies on this platform, in particular that the most suitable tasks are those that have a verifiable answer. Clearly those carrying out studies on crowdsourcing platforms must design robust experiments that do not rely on literacy, and due to its remote nature take measures to detect behavior that may undermine the integrity of the study. The user sample on Mechanical Turk was suitable as they are likely to be technology savvy adults. There were a number of study phases: *registration*: participants were requested to give information such as worker ID and demographic information; *enrolment*: where the participant would be given 30 seconds to view four key images randomly selected from our set of eight; *wait*: A JavaScript enforced stoppage of 30 minutes where participants could not progress to the next phase, but were free to carry out other tasks on Mechanical Turk. If participants attempted to progress beyond the wait period too quickly this could be detected via use of server-side timestamps. The last phase was *recall*: the participant attempts to recognize the images assigned to them and has a single attempt to do so.

Due to the remote nature of the study we designed the following study defenses in order to have increased confidence in our results: *anti-image caching*: the images presented at login were drawn from a different location on the server than those at enrolment. This removed the threat that the key images would load faster due to caching; *anti-print screen*: participation was restricted to Internet Explorer and via a JavaScript we cleared the

Figure 5: Example image grids assembled for key image #4 using similar, middle, and dissimilar decoy image criteria.

clipboard of the participant every 100 milliseconds. If consent to do this was not granted the experiment would not continue; *Dynamic key images*: key image sequences were not static across participants and there were ${}^{8}C_{4}$ different possibilities for the sequence of key images that could be presented. This meant that if images were recorded they may not be immediately reusable by another participant. *HTTP GET* parameter protection ensured that we could detect where parameters were maliciously altered in the browser or the back button was pressed

5.2 Results

5.2.1 Participation

We received 364 completed logins across a 6 day period. We treated as outliers those who completed the login procedure identifying no key images in less than 5 seconds. This reduced the numbers down to 343 with 117 in the similar condition, 112 in the dissimilar condition and 114 in the middle condition. In terms of demographics, 72% of participants were from India, with the United States the next prominent location at 7%. Most of the participants were male (73%). In terms of age, 269 were in the age group 18-30, 67 in the group 31-40, 15 in the age group 41-50, and 13 were 51+ years of age.

Table 3: The number of key images correctly identified (out of four) in study two. Success is 4/4.

Condition	Score Distribution				
	0	1	2	3	Success
Similar (n=117)	6 (5%)	14 (12%)	26 (22%)	37 (32%)	34 (29%)
Middle (n=114)	5 (4%)	8 (7%)	14 (12%)	20 (18%)	67 (59%)
Dissimilar (n=112)	0 (0%)	6 (5%)	6 (5%)	8 (7%)	74 (70%)

5.2.2 Accuracy

We firstly calculated a login success rate on a per-participant basis i.e. to compare the participants who correctly identified all 4 images. This was calculated by (successful logins/total logins). The raw data comprised success/fail value to represent a login. There was a significant difference between the performance of participants in the dissimilar group (70%) and the similar group (29%) $X^2(1,N=229)=37.716$, $p<0.01$. In addition there was a significant difference between the success rate in the middle (59%) and similar condition $X^2(1,N=231)=20.716$, $p<0.01$. The difference between the dissimilar and the middle condition was not statistically significant. Table 3 presents a more detailed illustration of participant performance. We also calculated a per-click success rate that represented (correct clicks/total clicks) for each condition. The benefit of this calculation is that it can give insight into accuracy in a manner less sensitive to a single mistake

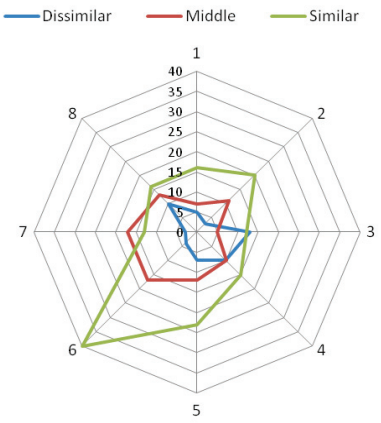


Figure 6: The number of login errors made per key image and per experimental condition.

by a participant. For example, a single problematic image grid could reduce login success rates considerably, whereas in reality this would reduce the per click success rate less severely. There was a significant difference between the success rates in the dissimilar condition (90%) and the similar condition (67%) $X^2(1,N=916)=57.679$, $p<0.01$. There was also a significant difference between the success in the middle (80%) and the similar condition, $X^2(1,N=924)=19.758$, $p<0.01$. The difference between the dissimilar and the middle condition using this metric was not significant.

Figure 6 illustrates the number of errors made per condition per image. Overall the errors do follow an intuitive pattern, they increase as the decoy images become more similar to the key image. However there are two exceptions, image 3 and image 7. Looking closely at the grids for these images, it is likely these errors can be explained by *inter grid key-decoy* similarity (see Section 3.1): a decoy image was visually similar to a non-collocated key image. This likely created confusion as to which image the user should select. This could indicate that the threshold we imposed on this instance of similarity was not sufficiently high. The graph also illustrates the interesting case of image 6 in the similar condition: there was a large number of user errors recorded when they were asked to identify this image. The decoy images for this image appeared visually and semantically similar. Analysis of errors made on a per-image basis across conditions highlighted a number of significant results too (see Table 4).

Table 4: Significant differences noted in user performance across experiment conditions on a per-image basis.

Image	Success %	Success %	X^2
	Similar	*Dissimilar*	
1	71%	91%	$X^2(1,120)=5.729$, $p<0.05$
2	66%	95%	$X^2(1,119)=14.540$, $p<0.01$
5	55%	87%	$X^2(1,105)=14.207$, $p<0.01$
6	32%	93%	$X^2(1,117)=46.230$, $p<0.01$
7	77%	95%	$X^2(1,81)=31.777$, $p<0.01$
	Similar	*Middle*	
5	55%	83%	$X^2(1,116)=10.449$, $p<0.01$
6	32%	70%	$X^2(1,116)=16.726$, $p<0.01$
	Middle	*Dissimilar*	
2	74%	95%	$X^2(1,101)=8.614$, $p<0.01$
6	70%	92%	$X^2(1,115)=10.125$, $p<0.01$
7	73%	95%	$X^2(1,123)=10.908$, $p<0.01$

5.2.3 Login Duration

We also recorded time required for users to login in each condition. This was recorded from the first grid appearing on-screen, until the final click. We treated the data as non-parametric due to the existence of a number of particularly long login durations distorting the mean. The median login duration was 57 seconds in the similar group, in the middle group 40 seconds, and in the dissimilar group 36 seconds. The difference between the similar and dissimilar conditions was significant in a Mann-Whitney U test ($Z=-4.730$, $p < 0.01$). Using a Wilcoxon 1-sample sign test we estimated the 95% confidence interval for the medians. This estimates that participants in the dissimilar condition would take between 33-40 seconds, in the middle condition 38-51 seconds, and in the similar condition 48-68 seconds. This suggests that the choice of decoy selection method could also have a significant impact upon the login durations. Figure 7 shows the distribution of login durations recorded for successful logins for each condition.

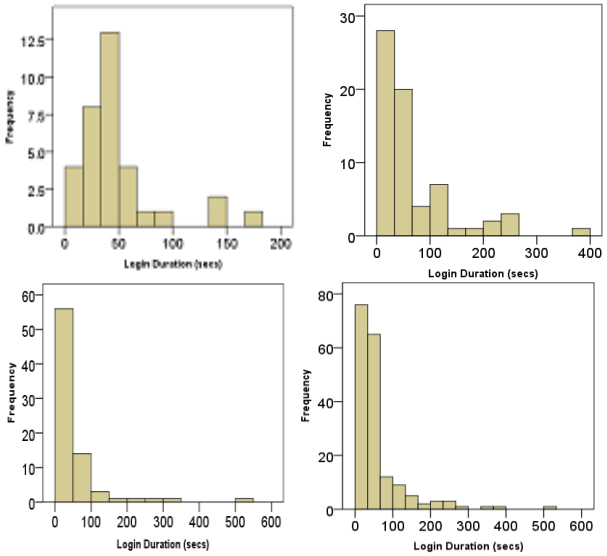


Figure 7: Length of successful logins in each condition: top left) similar; top right) middle; bottom left) dissimilar; bottom right) overall.

6. SECURITY IMPLICATIONS

The introduction of deliberate and measurable differences in visual similarity between images creates the potential for traces of this process to be left behind and exploited by attackers, who could infer key images and gain unauthorized access to systems. The particular threat is guessability, that patterns in the composition of the login challenge could be reverse engineered to allow an attacker to make better than random guesses. The goal

for an attacker is to obtain a successful login without any interaction with the user through activities such as coercion or observation attack. *Intersection attack* would allow a similar attack vector if unsecured [5][8]. We should assume that an attacker knows the method used for image filtering and any thresholds employed, and is able to capture the login challenge specific to a particular user. If the system provides *direct authentication* [33] we assume the attacker has compromised the username of the legitimate user and can capture the login images for offline analysis. If the infrastructure is *local authentication* then an attacker may be able to take a high quality photograph of the images, although this could be considered less likely. Other threats to be considered when introducing visual differences in the image filtering procedure include observation attack and description [9].

6.1 Key Relative Filtering

Key relative filtering [27] has been proposed in previous work as a method that is suitable for small image sets, as it imposes few constraints upon the login challenge composition. The procedure is to firstly identify the key image, reject all images within a similarity distance *d* of the key image and select decoy images randomly from the remaining images. However, an attacker could identify candidate key images even without knowledge of the thresholds being used, by recalculating pairwise similarities to search for patterns. One way to do this is to create a *similarity matrix* (see Figure 8) which is a simple visualization that captures pairwise EMD distances between images in a single *nxn* grid. In Figure 8 each large square represents the location of a single image, and the smaller squares within contain the EMD distance between that image and every other image in the 3x3 grid. The figure represents a particularly vulnerable case where the key relative similarity threshold is $d<3$. In this case the attacker could conclude that the centermost image has a good chance of being the key image, since it is the only image that exhibits such a careful pattern (d>3) in pairwise similarity values. Even without knowledge of the threshold the attacker could make a good guess at its value based upon the minimal *d* observed for each image in the matrix.

6.2 Exhaustive filtering

The analysis of key relative filtering has shown that for purposes of security a more holistic approach to image filtering should be taken, in order to hide traces of the filtering procedure. One approach is based upon ensuring a *minimum distance* exists between all images in the grid. One limitation of this approach is that while it enforces a minimal distance between all images, there is no upper bound, which could leave the login vulnerable to observation attack, as images exhibiting large visual differences could remain in the login challenge. An alternative approach that could eliminate this threat is based upon *similarity intervals,* where additionally an upper bound of similarity is also enforced. However, this approach would likely be difficult to implement in small image collections, as a greater number of images are likely to be rejected due to the increased number of similarity constraints upon a permissible image. There is a trade-off between the volume of images rejected in the filtering procedure and the number of constraints that are enforced. As a compromise, a *minimum distance* approach is likely to be suitable in smaller image sets and where observation attack or description is less likely to be a threat. An example of the visual differences that may result is illustrated in Figure 9. In order to minimize the number of images that must be rejected, a useful strategy in general involves:

1. Choosing a strategy for decoy selection i.e. similarity or dissimilarity.
2. Choosing a candidate key image, and calculating the distribution of pairwise similarity between it and the rest of the image set.
3. Sorting the images in ascending order of EMD, then, if choosing for dissimilarity, choosing from the back of the list, and if choosing for similarity, choosing from the front of the list.
4. Repeating 2-4 for each key image.

-	1.8	3.1	1.8	-	2.3	3.1	2.3	-
4.0	3.2	0.8	4.0	3.1	5.0	2.0	3.3	1.7
1.3	3.0	6.0	2.9	4.9	3.8	4.0	5.0	5.0
4.0	4.0	2.0	3.2	3.1	3.3	0.8	5.0	1.7
-	3.5	3.0	3.5	-	3.0	3.0	3.0	-
0.6	4.1	1.4	3.2	3.4	4.0	3.0	1.9	5.0
1.3	2.9	4.0	3.0	4.9	5.0	6.0	3.8	5.0
0.6	3.2	3.0	4.1	3.4	1.9	1.4	4.0	5.0
-	4.0	5.0	4.0	-	4.0	5.0	4.0	-

Figure 8: *Similarity matrix* that illustrates the pairwise EMD distance *d* between images in a single 3x3 image grid. This grid has been assembled with key relative filtering [27].

7. DISCUSSION

The vision of this research is that a system can take an arbitrary set of images and perform a filtering operation to generate a usable and secure login challenge, or else conclude that an image set does not contain suitable images for this purpose. Such a spontaneous approach to the generation of a login challenge becomes more useful when considering deployment level phenomena such as password resets, where ineffective recycling of images could cause confusion between new key images and old. A perfect automated semantic separation of images appears to be a difficult goal; however, we have shown that taking a coarse grained approach to the problem can affect usability. As a result it seems possible that ensuring a specific visual difference between images using pixel-level image signatures could assist in automated image selection strategies for recognition-based graphical passwords.

The recall test results suggest that comparison of pixel-level image signatures can affect the usability of recognition-based graphical passwords in terms of both user accuracy and the time required to login. We observed significant accuracy results between the similar and dissimilar condition, and the similar and middle condition. We did not observe significant differences between the middle and dissimilar condition. The login durations were also significantly impacted between the similar and dissimilar group with a significant difference of 21 seconds in the medians. The most damaging type of similarity we noted was inter grid key-decoy similarity, which is the similarity between a decoy image and a non-collocated key image. In this case the user erroneously selects a decoy image that appears similar to a non

Figure 9: Left) grids assembled using *minimum distance* approach where d=2; right) *similarity intervals* where 4>d>0.

collocated key image. Particularly conservative thresholds should be employed when considering this type of similarity. The remote scenario places success rates in a realistic zone, as participants were not within the sphere of influence of experimenters. Such results have important implications for deployment of recognition-based graphical passwords, as they serve to highlight the impact that seemingly subtle image choices can have upon the usability of the system.

We also identified the security risks inherent in automated image filtering where this is not performed holistically. The approach to exhaustive filtering based upon *similarity intervals* appears to be the most secure approach, resistant to reverse engineering of the similarity procedure and likely to be useful against observation attacks. However, the benefits must be traded off against the size of the required image set. Questions of an optimal image choice appear to be a pressing question for this genre of graphical passwords. However, in order to determine any benefits had by one image type or the other, it is necessary to carry out longitudinal studies that compare image types with differing levels of entropy and similarity involved. Recent work has compared objects and faces and questioned the superiority of face images [15].

The results have implications for graphical password systems of all genres. This work has focused upon recognition-based graphical passwords where there are multiple grids and one key image on each screen. However, our taxonomy of image filtering is relevant to systems where images are static for all users, or for configurations where key images are randomly distributed across the grids [5,13]. The results also have implications for other systems such as Passpoints [41], where future research could focus upon some notion of similarity between images to predict whether similar user choices could be expected between a number of images. Recent work correctly asserts that alternatives to alphanumeric passwords have so far failed to make an impact [14], juxtaposing novel mechanisms with alphanumeric passwords or PINs in terms of flexibility and convenience usually results in failure. The challenge for alternative authentication solutions is to demonstrate evidence of new types of value that impact the trade-off between security, usability, deployability, and convenience. In recent work [23] we made initial investigations how image similarity could impact observation attack and description in the context of a novel recognition-based graphical password system. Future work exploring automated image choice could help graphical passwords to provide such added value.

8. STUDY LIMITATIONS

In this study we did not consider the longitudinal memory impact of the recall task; we chose to model a short-term memory task as password enrolment is a particularly traumatic period for committing new credentials to memory. However, we believe our scenario introduced sufficient stress into the enrolment procedure to enable us to have confidence in the results. The success rates are constrained by the fact that users were not working with their own images and only had one attempt to identify the images. In addition, the participants were not logging into a real system and so not authenticating to access anything of value. Finally, the image processing intervention we chose only operated at the pixel-level. Study of more sophisticated techniques that incorporate object segmentation may prove to be a fruitful future research direction.

9. CONCLUSION

In this paper we explored the extent to which the process of assembling a usable login challenge for recognition-based graphical passwords based upon photographs could be automated using image processing techniques. In our tests we found that using a color histogram as an image signature and computing the distance between those histograms using the *Earth Mover's Distance* [30] provided a useful approach. In a short-term recall test with more than 300 people using Amazon Mechanical Turk, we found that automated choice of decoys to differing levels of visual similarity could impact the number of errors that users made at login, along with the login durations. We found significantly fewer errors made by users viewing grids with progressively dissimilar decoys compared to those viewing the most similar decoys. In the most significant case, we found that our automated decoy selection method could affect login success rates by 40%. This study illustrates that the performance benefits of recognition-based graphical passwords can be closely related to the chosen image sets.

10. ACKNOWLEDGMENTS

This work was funded by the UK Digital Economy Research Hub SiDE: Social Inclusion through the Digital Economy (EP/G066019/1).

11. REFERENCES

1. De Angeli, A., Coutts, M., Coventry, L., Johnson, G.I., Cameron, D., and Fischer, M.H. VIP: a visual approach to user authentication. *Proceedings of the Working Conference on Advanced Visual Interfaces*, ACM (2002), 316-323.
2. Biddle, R., Chiasson, S., and van Oorschot, P. Graphical Passwords: Learning from the first twelve years. *ACM Computing Surveys 44*, (2011).
3. Brady, T.F., Konkle, T., Alvarez, G.A., and Oliva, A. Visual long-term memory has a massive storage capacity for object details. *Proceedings of the National Academy of Sciences*, (2008).
4. Brostoff, S. and Sasse, M.A. Are Passfaces more usable than passwords? A field trial investigation. *Computer pages*, (2000), 405-424.
5. Dhamija, R. and Perrig, A. Deja Vu: a user study using images for authentication. *Proceedings of the 9th conference on USENIX Security Symposium - Volume 9*, USENIX Association (2000), 4.
6. Dirik, A.E., Memon, N., and Birget, J.-C. Modeling user choice in the PassPoints graphical password scheme. *Proceedings of the 3rd symposium on Usable privacy and security*, ACM (2007), 20-28.

7. Duncan, J. and Humphreys, G.W. Visual search and stimulus similarity. *Psychological review 96*, 3 (1989), 433-458.
8. Dunphy, P., Heiner, A.P., and Asokan, N. A closer look at recognition-based graphical passwords on mobile devices. *Proceedings of the Sixth Symposium on Usable Privacy and Security*, ACM (2010), 3:1--3:12.
9. Dunphy, P., Nicholson, J., and Olivier, P. Securing passfaces for description. *Proceedings of the 4th symposium on Usable privacy and security - SOUPS '08*, ACM Press (2008), 24.
10. Everitt, K.M., Bragin, T., Fogarty, J., and Kohno, T. A comprehensive study of frequency, interference, and training of multiple graphical passwords. *Proceedings of the 27th international conference on Human factors in computing systems*, ACM (2009), 889-898.
11. Ford, A. and Roberts, A. Colour Space Conversions. *University of Westminster Technical Report*, (1998).
12. Gudivada, V.N. and Raghavan, V.V. Content based image retrieval systems. *Computer 28*, 9 (1995), 18-22.
13. Hayashi, E., Dhamija, R., Christin, N., and Perrig, A. Use Your Illusion: secure authentication usable anywhere. *Proceedings of the 4th symposium on Usable privacy and security*, ACM (2008), 35-45.
14. Herley, C. and Oorschot, P.V. A Research Agenda Acknowledging the Persistence of Passwords. *IEEE Security and Privacy 99*, PrePrints (2011).
15. Hlywa, M., Biddle, R., and Patrick, A.S. Facing the facts about image type in recognition-based graphical passwords. *Proceedings of the 27th Annual Computer Security Applications Conference*, ACM (2011), 149-158.
16. Kittur, A., Chi, E.H., and Suh, B. Crowdsourcing user studies with Mechanical Turk. *Proceedings of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, ACM (2008), 453-456.
17. Klein, D.V. ``Foiling the Cracker'' -- A Survey of, and Improvements to, Password Security. *Proceedings of the second {USENIX} Workshop on Security*, (1990), 5-14.
18. Komanduri, S. and Hutchings, D.R. Order and entropy in picture passwords. *Proceedings of graphics interface 2008*, Canadian Information Processing Society (2008), 115-122.
19. Kuo, C., Romanosky, S., and Cranor, L.F. Human selection of mnemonic phrase-based passwords. *Proceedings of the second symposium on Usable privacy and security*, ACM (2006), 67-78.
20. Lee, S.M., Xin, J.H., and Westland, S. Evaluation of image similarity by histogram intersection. *Color Research & Application 30*, 4 (2005), 265-274.
21. Lv, Q., Charikar, M., and Li, K. Image similarity search with compact data structures. *CIKM '04: Proceedings of the thirteenth ACM international conference on Information and knowledge management*, ACM (2004), 208-217.
22. Medin, D.L. Concepts and Conceptual Structure. *American Psychologist 44*, 12 (1989), 1469-1481.
23. Nicholson, J., Dunphy, P., Coventry, L., Briggs, P., and Olivier, P. A security assessment of tiles: a new portfolio-based graphical authentication system. *Proceedings of the 2012 ACM annual conference extended abstracts on Human Factors in Computing Systems Extended Abstracts*, ACM (2012), 1967-1972.
24. Pass, G. and Zabih, R. Histogram refinement for content-based image retrieval. *Applications of Computer Vision, 1996. WACV '96., Proceedings 3rd IEEE Workshop on*, (1996), 96-102.
25. Passfaces Corporation. *The Science Behind Passfaces*. .
26. Pering, T., Sundar, M., Light, J., and Want, R. Photographic Authentication through Untrusted Terminals. *IEEE Pervasive Computing 2*, 1 (2003), 30-36.
27. Renaud, K. and Olsen, E.S. DynaHand: Observation-resistant recognition-based web authentication. *Technology and Society Magazine, IEEE 26*, 2 (2007), 22-31.
28. Renaud, K. On user involvement in production of images used in visual authentication. *J. Vis. Lang. Comput. 20*, 1 (2009), 1-15.
29. Rubner, Y., Tomasi, C., and Guibas, L.J. A Metric for Distributions with Applications to Image Databases. *Proceedings of the Sixth International Conference on Computer Vision*, IEEE Computer Society (1998), 59--.
30. Rubner, Y., Tomasi, C., and Guibas, L.J. The Earth Mover's Distance as a Metric for Image Retrieval. *Int. J. Comput. Vision 40*, 2 (2000), 99-121.
31. Salehi-Abari, A., Thorpe, J., and Oorschot, P.C. van. On Purely Automated Attacks and Click-Based Graphical Passwords. *Proceedings of the 2008 Annual Computer Security Applications Conference*, IEEE Computer Society (2008), 111-120.
32. Shepard, R.N. Recognition memory for words, sentences, and pictures. *Journal of Verbal Learning and Verbal Behavior 6*, 1 (1967), 156-163.
33. Smith, R.E. *Authentication: From Passwords to Public Keys*. Addison Wesley, 2001.
34. Smith, T.A., Jones, L.V., and Thomas, S. Effects upon verbal learning of stimulus similarity, number of stimuli per response, and concept formation. *Journal of Verbal Learning and Verbal Behavior 1*, 6 (1963), 470-476.
35. Squire, D.M. Learning a similarity-based distance measure for image database organization from human partitionings of an image set. *Applications of Computer Vision, 1998. WACV '98. Proceedings., Fourth IEEE Workshop on*, (1998), 88-93.
36. Suo X., Z.Y.O.G.S. Graphical Passwords: A Survey. *Proceedings of the 21st Annual Computer Security Applications Conference*, IEEE Computer Society (2005), 463-472.
37. Tullis, T.S., Tedesco, D.P., and McCaffrey, K.E. Can users remember their pictorial passwords six years later. *Proceedings of the 2011 annual conference extended abstracts on Human factors in computing systems*, ACM (2011), 1789-1794.
38. Tullis, T.S. and Tedesco, D.P. Using personal photos as pictorial passwords. *CHI '05 extended abstracts on Human factors in computing systems*, ACM (2005), 1841-1844.
39. Wang, J.Z., Li, J., and Wiederhold, G. SIMPLIcity: Semantics-Sensitive Integrated Matching for Picture LIbraries. *IEEE Trans. Pattern Anal. Mach. Intell. 23*, 9 (2001), 947-963.
40. Weinshall, D. Cognitive authentication schemes safe against spyware. *Security and Privacy, 2006 IEEE Symposium on*, (2006), 6 pp. -300.
41. Wiedenbeck, S., Waters, J., Birget, J.-C., Brodskiy, A., and Memon, N. PassPoints: design and longitudinal evaluation of a graphical password system. *Int. J. Hum.-Comput. Stud. 63*, 1-2 (2005), 102-127.
42. Yee, H. PerceptualDiff. http://pdiff.sourceforge.net/.
43. Zurko, M.E. and Simon, R.T. User-centered security. *Proceedings of the 1996 workshop on New security paradigms NSPW 96*, ACM Press (1996), 27-33.

Building Better Passwords using Probabilistic Techniques

Shiva Houshmand
Florida State University
Department of Computer Science
Tallahassee, FL 32306-4530
1 (850) 645-7339
sh09r@my.fsu.edu

Sudhir Aggarwal
Florida State University
Department of Computer Science
Tallahassee, FL 32306-4530
1 (850) 644-0164
sudhir@cs.fsu.edu

ABSTRACT
Password creation policies attempt to help users generate strong passwords but are generally not very effective and tend to frustrate users. The most popular policies are rule based which have been shown to have clear limitations. In this paper we consider a new approach that we term *analyze-modify* that ensures strong user passwords while maintaining usability. In our approach we develop a software system called AMP that first analyzes whether a user proposed password is weak or strong by estimating the probability of the password being cracked. AMP then modifies the password slightly (to maintain usability) if it is weak to create a strengthened password. We are able to estimate the strength of the password appropriately since we use a probabilistic password cracking system and associated probabilistic context-free grammar to model a realistic distribution of user passwords. In our experiments we were able to distinguish strong passwords from weak ones with an error rate of 1.43%. In one of a series of experiments, our analyze-modify system was able to strengthen a set of weak passwords, of which 53% could be easily cracked to a set of strong passwords of which only 0.27% could be cracked with only a slight modification to the passwords. In our work, we also show how to compute and use various entropy measures from the grammar and show that our system remains effective with continued use through a dynamic updating capability.

Categories and Subject Descriptors
K.6.5 [**Management of Computing and Information Systems**]: Security and Protection – *authentication*; K.4.4 [**Computers and Society**]: Electronic Commerce—*security*

General Terms
Security, Design, Experimentation, Human Factors

Keywords
Password checking, Password creation policies, Information security, Strong authentication

1. INTRODUCTION
Internet based systems such as online banking and online commerce continue to rely heavily on passwords for authentication security. Human memorable passwords are thus a key element in the security of such systems. However, most users do not have the information to ensure that they are in fact using a "strong" password rather than one that can easily be broken. This limitation has led to the use and advocacy of *password creation policies* that purport to help the user in ensuring that the user chosen password is not easily breakable. The most prevalent password creation policy is the *rule-based approach* wherein users are given rules such as minimum length of eight characters and must contain an upper case letter and a special symbol. It has been shown by several authors that this approach by itself is not very effective [1, 2]. A second type of password creation policy can be termed the *random approach* where an effectively random string is given by a system to the user. Clearly the random approach has the problem that the given string is in general non-memorable so the purpose of having a password that can easily be remembered is defeated.

We advocate a third approach which is to have a system that *analyzes* a user proposed password and then *modifies* it if it is weak to make it strong, yet preserves enough of the original password so that the new password is still memorable. We do this by changing the original password by only an edit distance of one. We term this approach *analyze-modify*. It can be viewed as having a reject function that rejects a weak password and then a modify function that changes that weak password somewhat to one which is appropriately strong.

We show that empirical analysis based on trying to crack passwords using probabilistic techniques [3] can be adapted to analyzing the strength of passwords. We show how the associated probabilistic context-free grammar can be used to build a realistic reject function that can distinguish between strong and weak passwords based on a threshold probability. An obvious component of an empirical analysis might be to have a dictionary of popular passwords and ensure that the modified password is not one of these. But a more important consideration is to show that the modified password is not likely to be cracked using any technique as we are able to do. The black-listing approach is automatically subsumed by our approach simply by the choice of dictionaries. Note that we are interested in protecting against off-line attacks where an attacker has obtained a set of hashes (and likely user names) and desires to break as many passwords as possible in a reasonable amount of time.

We illustrate the effectiveness of our prototype system, called Analyzer and Modifier for Passwords (AMP), through a series of experiments on three lists of disclosed passwords. AMP was able to modify weak passwords and strengthen them to strong passwords that were within an edit distance of one from the user passwords. Using the popular password cracker John the Ripper [4], AMP-designated weak passwords were broken at levels of 48%, 39% and 53% respectively from the three lists. The strengthened passwords could only be broken at levels of 0.10%, 0.51% and 0.27% respectively from the three lists. We further develop a dynamic update system for AMP that continually changes how it proposes the modified passwords. We show that

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

continual use of our system will continue to generate strong passwords, even if attackers are aware of the probabilistic distribution of passwords. We discuss the use of various entropy measures that help to substantiate this claim.

In section 2 we discuss previous approaches to the problem of ensuring that users have strong and memorable passwords, classical entropy measures and the probabilistic password cracking system we use. In section 3 we develop our approach for analyzing the strength of passwords and in section 4 we discuss our password modification component. In section 5 we discuss the update system of AMP. In section 6 we illustrate the effectiveness of our system using attacks by two different password cracking systems on real user passwords collected from a variety of sites. We briefly discuss some future work and conclusions in section 7.

2. BACKGROUND & PREVIOUS WORK

2.1 Previous Work on Password Checking

Although not really an analysis of password strength, many studies attempt to determine various aspects of how users choose passwords. Riley [5] reports that in a study of 315 participants, about 75% of them reported that they have a set of predetermined passwords that they use frequently. Almost 60% reported that they don't change the complexity of their password depending on the nature of the website they use. Stone-Gross et al. [6] collected around 298 thousands passwords from the Torpig botnet. They found that almost 28% of users reused their passwords and they managed to crack over 40% of the passwords in less than 75 minutes. This illustrates that having strong passwords for less important websites such as social networking websites is likely to be as necessary as for websites such as online banking.

Most organizations and websites follow a rule-based approach in recommending or enforcing password policies. A study by Shay et al. [7] showed that users were not happy about changing the password creation policy to a stricter one and that it took on average 1.77 tries to create a new password accepted by the system based on a new password creation policy recently instituted. Riley [5] also reports that the average length of time users maintained their primary password was reported as 31 months and 52% of them never change their password at all.

Rule-based advice is confusing as there is no consistency across systems and websites in the requirements, with differing advice about length, number of symbols and digits, and even in the symbols that can be used. In [8] it is shown that inconsistent and even contradictory recommendations make such advice unreliable for users. The U.S. NIST guideline [9], the basis for most rule-based policies, proposed a rule-based approach that used the notion of Shannon entropy for estimating password strength based on suggested values of the components of the password. However, Weir et al. [1] performed password cracking attacks against multiple sets of real life passwords and showed that the use of Shannon entropy as defined in NIST is not an effective metric for gauging password strength and it does not give a sufficient model to decide on the strength of a given password.

Password expiration policies are designed to ensure stronger passwords over time. However, Zhang et al. [10] showed that an attacker can easily get access to an account by capturing the account's previous passwords. They suggest that at least 41% of passwords can be broken offline from a previous password in a matter of seconds and only five online password guesses suffices to break 17% of accounts. A more recent study [11] reports that although nowadays users understand the importance of secure behavior, they still find it too difficult to cope with password creation policies, and they rarely change their passwords due to the frustration of creating a new password along with the difficulty of memorizing it. In studies by Charoen et al. [12] and Adams and Sasse [13], it was found that users are not even unanimous about the necessity of having a strong password and the reason users choose insecure passwords is because they usually do not know how to create secure ones. Studies [14] show that even restrictive password creation policies do not have impact on the use of meaningful information in passwords, nor does it reduces reusing the password. Reuse can subject users to other types of attacks such as phishing, key-logging and targeted attacks [15]. A study by Shay et al. [16] shows that the more restrictive and complicated the policy, the less user-friendly it is.

There have been some studies [17, 18] exploring the use of the random password generation approach. The major problem is the usability of the password for the user since such a password has typically no context for the user and is naturally hard to remember. Forget et al. [19] studied the memorability of passwords by randomly inserting or replacing fixed number of characters in a user chosen password. They showed that once the users confirmed their changed passwords, they could recall it as easily as the control group (passwords without change). However, they did not develop a methodology for analyzing the strength of these passwords.

Generating secure passwords is a tradeoff between creating a password that is hard to crack and usable. Some studies of passwords [15, 18] try to provide an understanding of how various policy factors make creating passwords easier, memorable, and usable, but none of them seem to have been applied in practice.

The work by Verheul [2] is an excellent example of trying to understand the relationship of various entropy measures in order to build better passwords. Verheul showed how to build reasonable short secure passwords based on calculating the Shannon entropy with assumptions on the min entropy and guessing entropy. However, there was no attempt in this paper to consider the usability or memorability of the passwords or how to modify a user suggested password.

The analyze-modify approach also has some related history. The analysis is usually a simple way to determine if a password is weak such as checking against a dictionary. Note that in reality this is not really a sufficient condition for a password to be strong. Current proactive password checkers generally follow such a black-listing approach. See for example Yan [20] and Spafford [21]. However, simple black-listing approaches generally have problems with any sophisticated dictionary based attack.

Perhaps the most relevant study related to our approach is Schechter et al. [22] in a study on popularity of passwords. They propose to build an oracle for existing passwords that are available to the Internet-scale authentication systems. They recommend that such popular passwords be disallowed and the main thrust of their work is to devise a way to efficiently store the large number of popular passwords that would be prohibited. An open question posed in their study is how to use the oracle without revealing the actual password to attackers while querying online. We show that our technique gets around this problem as well as their storage problem. More recently [23] explores measuring the strength of passwords using a Markov approach.

Our approach was first hinted at in Weir et al. [1] where it was suggested that a probabilistic password attack system could be

used to determine if a proposed password was weak and should be rejected. We show in this paper how such a probabilistic cracking system can in fact be used for analyzing passwords. Once such an analysis is done, we then show how identified weak passwords can be effectively modified to be strong.

We next review some classical mathematical notions that have been proposed to measure strengths of passwords. After this, we review the probabilistic password cracking [3] approach that we use for doing the password analysis component of AMP.

2.2 Classical Measures of Password Strength

Entropy as a way to measure the uncertainty arising from a probability distribution was suggested by Claude Shannon [24] in an effort to explore the uncertainty of letters and words in English.

Definition 2.1 Shannon Entropy: Let X be a discrete random variable with probability mass function $p(x) = Pr\{X=x\}$, $x \in X$. The entropy $H(X)$ of such a random variable X is defined by:

$$H(X) = -\sum_x p(x) \log_2 p(x) \quad (2.1)$$

The notion of guessing entropy was introduced in [25].

Definition 2.2 Guessing Entropy: Assume that the probabilities p_i are denoted in a decreasing form $p_1 \geq p_2 \geq \ldots \geq p_n$. Then the guessing entropy denoted by $G(X)$ is:

$$G(X) = \sum_{i=1}^{i=n} i p_i \quad (2.2)$$

The idea here is that in an optimal attack, the attacker would try the highest probability passwords first and thus guessing entropy measures the expected number of tries until success. However, it was shown by Verheul [2] that simply having a high value of the guessing entropy itself is not sufficient since a distribution with a high value of guessing entropy is possible, even with the probability of the first potential guess p_1 being very high and thus easily broken. A third notion is thus often used which is called the min entropy ($H_\infty(X) = - log_2 p_1$) [2]. If the min entropy is high then the probability of the first password is small. At the time of much of this research, investigators really had no idea of the actual distribution of passwords or how to determine the above entropy values in any realistic setting. This began to change with hackers posting large numbers of revealed passwords on-line. An important development at this stage was the probabilistic password cracking work based on training a context-free grammar and using this grammar as an effective model to simulate an optimal password cracking attack (trying the highest probability passwords first).

2.3 Probabilistic Password Cracking

We used the password cracking system of Weir et al. [3] for our work. The authors used probabilistic context-free grammars to model the derivation of real user passwords and the way users create their passwords. The goal was to generate realistic guesses in decreasing order of probability where the probabilities are derived through training on large sets of revealed passwords. In [3] password string components consisting of alphabet symbols are denoted as L, digits as D, special characters as S and capitalization as M. The authors also associate a number to show the length of the substring. For example, the password "football123!$" would be $L_8D_3S_2$. Such strings are called the *base structures*. There are two steps in this password cracking approach [3]: the first is generating the context-free grammar from a training set of disclosed real user passwords and the second is generating the actual guesses in probabilistic order using the grammar.

2.3.1 Training

The observed base structures and their frequencies are derived from the training set of passwords. Information about the probability of the digits, special characters and capitalization (case) are also obtained from the training set. This information is used to generate the probabilistic context free grammar. The probability of any string derived from the start symbol is then the product of the probabilities of the productions used in its derivation. See Table 1. Using this grammar, for example, we can derive password "987dog!" with probability 0.04992.

$$\mathbf{S \rightarrow D_3L_3S_1 \rightarrow 987L_3S_1 \rightarrow 987dogS_1 \rightarrow 987dog!}$$

The learning phase does not actually include determining probabilities of the alphabet strings since these are not considered to be a sufficient sample even for large training sets. Instead, for example, the L_3 part of the guess comes from a dictionary with probability equal to one over the number of words of length 3. Furthermore, probability smoothing can be used to give an appropriately low probability value to digits, special symbols, case and base structures that do not arise in the training set.

2.3.2 Generating Password Guesses

The guess generation phase generates the possible password guesses in decreasing probability order using the context-free grammar obtained from the previous step. Note that this order is what we need to compute the guessing entropy. Multiple dictionaries can be used with probabilities associated to each.

Table 1. Example probabilistic CFG

Left Hand Side	Right Hand Side	Probability
S→	$D_3L_3S_1$	0.8
S→	S_2L_3	0.2
D_3→	123	0.76
D_3→	987	0.24
S_1→	!	0.52
S_1→	#	0.48
S_2→	**	0.62
S_2→	!@	0.21
S_2→	!!	0.17
L_3→	dog	0.5
L_3→	cat	0.5

3. ANALYZING PASSWORD STRENGTH

For a password to be strong we need to make sure that it cannot be easily broken. For memorability we start with the assumption that the original password chosen by the user is a memorable password for that particular user. Our first step is to evaluate the user chosen password for strength based on the probability of that password being able to be cracked. For this we use the probabilistic password cracking system trained on a comprehensive set of real passwords. In fact, we are able to determine a *threshold value* below which a password would be considered as strong. This allows us to build a *reject function* that will accept a strong password but reject a weak password. Weak passwords are then modified by AMP to make them strong. An overview of different components of the AMP system is illustrated in Figure 1. In the preprocessing phase, we train the system on real user passwords using the same technique used for training a probabilistic password cracker. This results in a probabilistic context free grammar that can generate guesses in highest probability order. We assume that the training set used in this step is a comprehensive set of passwords (and a sufficiently large sample

set) that can be used as a model of realistic passwords. Once we have this we need to determine the threshold value.

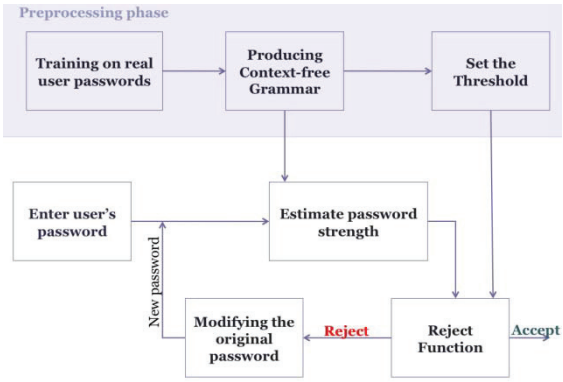


Figure 1. Flow chart of AMP design.

3.1 Setting the Threshold

We now turn to the specific issue of distinguishing between weak and strong passwords. A strong password is one for which it takes an attacker an appropriately long *cracking time* (*ct*) to crack that password (in hours). Our assumption is that in an online attack, the attacker would use the optimal strategy of trying the highest probability passwords in decreasing order of probability. We define the ***threshold value*** (*thp*) as that probability such that passwords with probability less than *thp* are strong and those that are greater than or equal to *thp* are weak. Because we use our probabilistic context-free grammar (plus appropriate dictionaries) as our model of the realistic password distribution, we can determine the number of guesses *g(thp)* the attacker would make before trying a password with a value equal to the threshold value *thp*. Let *r* be the rate-per-hour of the guesses (based on the hash type, cracking system speed, etc.). We thus have g(thp) = ct * r. We now need to determine thp given g(thp) since we will use this value to decide whether a given password is strong or weak. There are two ways that we can determine the threshold.

In the first approach, we run the probabilistic password cracker of the pre-processing phase (once) and generate a table that produces guesses and their probabilities at various time intervals. Although this approach is accurate and straightforward, it is not always feasible to reach the desired number of guesses due to time and resources as we may have to run our guesser for a very long time. Table 2 shows the threshold table produced by running a guess generator trained on a set of over 1 million passwords. If we set the threshold at 2.96×10^{-13} and the probability of a password is less than this threshold, we know that it will take at least 1 day to crack that password using an optimal password cracking strategy.

The second approach gives us only a lower bound for the number of guesses *g(thp)* until we reach a given value *thp* but it only requires using the context-free grammar and does not require actually generating all the guesses. Thus it is very fast. Although it only gives a bound, it is conservative with respect to ensuring that a proposed password is strong. The algorithm starts with a threshold *thp* and estimates the number of elements in each base structure *i* (with probability p_i) that are greater than this value. By doing a binary search we can find a password with an index in each component of the base structure whose probability is the closest one to thp/p_i, and thus calculate the number of guesses with probability greater than this value. We do this for each base structure. This will also give us a table of the probabilities with the associated total number of guesses such as Table 2.

Table 2. Thresholds for the training_pswd_checker grammar

Total number of guesses	Probability values	Time (in hours) (On 2.4GHz Intel Core 2 Duo, MD5 hash)
1,800,000,000	1.31×10^{-11}	1
14,400,000,000	1.59×10^{-12}	8
21,600,000,000	1.20×10^{-12}	12
28,800,000,000	6.37×10^{-13}	16
43,200,000,000	2.96×10^{-13}	24
86,400,000,000	9.94×10^{-14}	48
129,600,000,000	6.70×10^{-14}	72
172,800,000,000	5.29×10^{-14}	96
187,200,000,000	4.70×10^{-14}	104

3.2 The AMP Reject Function

AMP starts with asking users to enter their chosen password. Using the probabilistic context-free grammar of the pre-processing phase, we calculate the probability of the chosen password as follows: parse the given password to its components. For example if the password is *Alice123!,* we parse it to $L_5M_5D_3S_1$. Next, we find the probability of the base structure $L_5D_3S_1$ along with the probabilities of *alice*, *123*, *!* and the mask *ULLLL*. The product of these probabilities is the probability of the user's password. This probability p_u is compared with the threshold value to accept or reject the password. An issue that might occur in this phase is not being able to determine the probability of p_u from the context-free grammar. This could happen if the base structure or some other components of the user-chosen password is not included in the derived context-free grammar. Here is how we handle some of these situations for each component: If the base structure of the user-chosen password is not included in the context-free grammar, we can either assume that the password is strong enough and accept it, or we can find the lowest probability for the base structures and set it as an estimate for the probability of this base structure. We currently use this latter approach. If the digit component of the password or the special characters component were not initially in the training data, we can still find a probability associated with those values since our grammar includes these not-found values through smoothing. If the alphabet part of the password is not included in the dictionary, we use the probability associated with a word of the same length as in the dictionary, since we currently assume all the words of the same length as having equal probability. We have thus shown that we can determine if the user's password is strong or weak. We next discuss how to modify a weak password.

4. MODIFYING A WEAK PASSWORD

When AMP rejects a password we need to modify the password but we want to still keep it usable and memorable. A usable password is a password that is easy to remember and type. Things people can remember are different for each group of people based on their age, situation, location, etc. There are also special numbers or names that are only important to an individual. We believe users should be free to choose any word, numbers or special characters that make sense to them when initially proposing a password. (An exception is that the password be long enough as otherwise brute force would be enough to crack it.)

If a password is rejected we try to generate passwords with slight changes to the user-chosen password using the AMP distance function. This is based on edit distance to fulfill the need of usability for users. We believe users choose password components for memorability and only minimal changes should be made.

Hence, we start generating passwords with distance one from the user-chosen password and check if the modified password is within the acceptable threshold value. Forget et al. [19] showed that even several random changes (such as replacing or inserting up to four characters) were memorable by users. We can conclude from this study that our modifications, which usually change only *one* character in the passwords, will have a similar result. Note that in many cases (when the user chosen password is strong) we do not need to change the password at all.

4.1 The AMP Distance Function

We use a distance function similar to the Damerau-Levenshtein distance function [26] but with some changes to make it more efficient for our password structures. We define two sets of operations for our distance function. An operation can be done either on the base structure or on a component.

4.1.1 Operations on the Base Structure:

Insertion: inserting a component of length one is allowed only when it is not of the same type as its adjacent components. Example: if the base structure is $L_5D_3S_1$ we can insert D_1 in the beginning to make $D_1L_5D_3S_1$.

Deletion: deleting a component can be done if the number of components is not 1 and if it does not make two components of the same type adjacent. Example: we can delete D_2 from base structure $D_2S_1D_1$ to make S_1D_1.

Transposition: exchanging two adjacent components can be done only if it does not make two components of the same type adjacent to each other.

4.1.2 Operations on the Component:

Insertion: inserting one character of the same type inside a component is allowed. Example: if component D_3=123, we can transform it to 4123 by inserting 4 in the beginning.

Deletion: Deleting one character inside a component is allowed only if the length of the component is not equal to 1.

Substitution: we can substitute a character with another character of the same type. If S_2=!! it can be transformed into "!#".

Case: we can invert the case (upper / lower) of one character in the alphabetical component of the password. Example: If L_4 = life, we can transform to liFe.

The cost of each operation is 1 in our current system but we could allow for different costs for different operations if needed. Note that we do not make changes in the alpha string part except possibly to change case.

4.2 Modifier Algorithm

When users enter their password, the system automatically parses the password to its base structure. To construct possible modified passwords, we execute the above operations on the base structure and components and keep all the information in a tree structure. The user-chosen password is the root of the tree and we keep track of the results of all possible operations within edit distance of one. In the tree, a child node is the result of one of the operations. After building the tree, we start from the root node and randomly choose a child until we reach a leaf node. If this password is within the acceptable threshold limit we are done, else we randomly try again. During each tree traversal, we actually mark each component previously tried so that the next pass will find a different password.

Table 3. Example of passwords modified by AMP

Input password to AMP	Output of modifier
trans2	%trans2
colton00	8colton00
789pine	789pinE
mitch8202	mitch=8202
cal1fero	cal8fero
KILLER456	KILlER456
violin22	violin^22
ATENAS0511	0511AETENAS
*zalena6	*3zalena6
KYTTY023	KYTTY023r

It is possible that we will not find a password within distance 1 with the desired probability. By selecting the new passwords randomly, we are avoiding the possibility of suggesting the same password to different users with the same or similar original passwords. In order to get a password with distance 2, one could repeat the same steps for passwords with distance 1 starting from any of the distance 1 passwords. Table 3 shows a set of passwords given to AMP as input and the output of the modifier component. It can be seen that very limited changes has been applied to the user-chosen password thus preserving the usability and memorability of the password.

5. DYNAMIC UPDATE

In order to maintain the AMP system as still effective after users use the system for some time, we have developed an update strategy that modifies the grammar periodically. AMP clearly proposes less popular passwords to users (those having smaller probabilities) than the common ones they might suggest when modification is needed. However, what could happen after using the system for a period of time is that the probability distribution of passwords changes due to the passwords proposed by AMP. And therefore, whenever a recent set of real user passwords become revealed, an attacker can use these for training the password cracker. Since the supposedly strong passwords suggested by AMP have become in use more often and would now have higher probability in the guessing generator, the attacker has a better model of the AMP generator and therefore continued use of the original grammar could be problematic. Obviously, we can always use the most recent set of passwords as the training set for AMP to overcome this problem, but it would not always be easy to access a large set of real-user passwords. Instead, we consider every modified password that has been suggested to a user as a publicly disclosed password, with an appropriate weight, to be used as if it were in the training set. By effectively adding every new password to the training set, we always have a realistic and recent probability distribution for the probabilistic grammar. For example, if some password structure has low probability and thus AMP keeps suggesting that structure, after a while, since we are adding every password to the training set, AMP will dynamically adapt and use that structure less frequently.

5.1 Updating the Grammar

Note that in order to update the training set, we do not actually need to add the AMP proposed password to the training set, repeat the training step and reproduce the context-free grammar again. Instead, we only need to adjust the probability values in the context-free grammar. The probability values in the grammar are the frequencies of each structure or component used in the

training set. Whenever a new password has been suggested we only need to update the frequency of the components and base structures used in that password. For example, if the new password is *!!78liar*, we only change the probabilities of the base structure and of S_2 and D_2. We do not change the probability of *liar* since as previously discussed probabilities of words do not come from the training set but from the dictionary and AMP considers all the words (whether they are included in the dictionary or whether they are not) the same based on length.

By considering the probability of each element (of the base structure or the component) as its frequency in the training set we have $p_i = n_i / N$, where n_i is the number of occurrences of the element and N is the total number of elements. With this in mind, seeing another element i would change the probability to $p_i' = (n_i+\alpha)/(N+\alpha)$ and the probability of the rest of the elements would change to $p_j' = n_j / (N+\alpha)$. The parameter α can be used to adjust the rate of change. This mechanism is similar to Laplacian smoothing. In our experiments we trained our grammar on approximately 1 million passwords which resulted in about 11 thousand base structures. Updating this grammar can be done almost instantaneously. Every time we update the grammar, some of the probability values change and it changes the password distribution. In order to be able to understand the changes and compare the distributions we use entropy metrics to see how the dynamic update affects the probabilities. First we explain how these metrics can be relevant, then we show our approach to calculate Shannon entropy from the grammar and then we discuss the effect of updating the grammar with respect to the entropy values.

5.2 Using the Entropy Metrics

So how can we use the entropy measures? We need to distinguish between problems with theoretical distributions and empirical distributions of passwords. For empirical distributions we can first check if the min entropy is low. This is the same as the probability of the first guess being high. This simply means that a few initial passwords might easily be guessed. But this could be expected in any realistic distribution. In general in our experiments we have seen that the min entropy is actually fairly high. But in fact the min entropy is really irrelevant to us since we would never propose the first few high probability passwords anyways. We can also check if the Shannon entropy distribution is reasonably high. Because we have a quick way to compute the exact Shannon entropy $H(X)$ of our probability distribution (see next section), we also have a quick way to compute a lower bound on the guessing entropy G, using a bound derived by Massey [25].

$$G(X) \geq ¼\ 2^{H(X)} + 1 \qquad (5.1)$$

In the experiments in Section 6, the Shannon entropy of the original password checker distribution is about 27, which can be viewed as equivalent to a space of 2^{27} different passwords. Note that even with the moderate value of Shannon entropy, the total number of guesses possible by the grammar is beyond the trillions. Thus we should be able to find a reasonable reject function as there will be many possible passwords with very small probabilities. Finally, if we can drive the system to higher Shannon entropy then the new distribution of passwords would be more resistant to any password cracking attack. Thus, used properly, Shannon entropy of the grammar can be useful in our grammar update system.

It should be noted that when ensuring strong passwords, there are two possible approaches. The first is to find a distribution from which *any* password chosen is hard to break. This is the approach taken by Verheul [2]. Thus the guessing entropy has some meaning but as discussed by Verheul, he needs to also ensure that the min entropy is high. Recall that Verheul's approach does not ensure usability of the chosen password. The second approach is to somehow ensure that a specific password is hard to break, but it need not be randomly chosen from a given distribution. This is our approach and we additionally ensure usability. Furthermore we protect against an optimal guessing attack. But we clearly do not care that some initial number of passwords from that distribution can be broken since these would be identified as weak by our system.

5.3 Shannon Entropy of the Grammar

Since we have a password guess generator that can generate passwords in probabilistic order, we can clearly compute the Shannon entropy, guessing entropy and min entropy for the guesses generated by our context-free grammar by generating guesses and computing the various entropy values. Note that since the generator was developed through training on revealed passwords, these entropy values can be viewed as realistic values for the relevant password probability distribution. In fact after training on a sufficiently large set of revealed passwords, the distribution (through the grammar) can be viewed as a reference model for the "true" probability distribution of passwords.

We now show that we can compute the Shannon entropy using only the probabilistic-context free grammar and without actually generating these guesses. We use some well-known notions such as joint entropy and conditional entropy [27]. Let G to be the random variable that is the distribution of the strings which are the passwords derived from the grammar. More precisely, it is the distribution of derivation trees but since the grammar is non-ambiguous it can be viewed as the strings themselves. The context-free grammar for a password distribution can be viewed as composed of several distributions. One is from the start symbol S to the *base structures*, which we call the random variable B, and the second one is from the sentential forms of base structures to the terminals (the password guesses) which we call the random variable R (for *rest*). For example, if we have base structures that can take n different values $b_1, b_2, \ldots, b_n$ where n is the total number of base structures, then since $S \rightarrow b_i$ then we have $p(S \rightarrow b_i) = p(B=b_i)$. Note also that the random variable R | B is itself computable from the probability distributions represented by each component of the base structure because of independence of the component derivations. Table 4 is a simple example of the context free grammar of Table 1 illustrating B and $R \mid B$ with some sample probability values.

Table 4. Example CFG for entropy calculation

Random Variable B		Random Variable R | B	
Rule	**Probability**	**Rule**	**Probability**
$S \rightarrow D_3L_3S_1$	0.8	$D_3L_3S_1 \rightarrow$ 123dog!	0.1976
		123dog#	0.1824
		123cat!	0.1976
		987cat#	0.0576
$S \rightarrow S_2L_3$	0.2	$S_2L_3 \rightarrow$ **cat	0.31
		!!dog	0.085

Not shown are the random variables L_5, D_3, S_1, L_3, and D_2. For example, the random variable D_3 has the distribution as shown in Table 1. We have derived the following theorem that can be used to calculate Shannon entropy from a context-free grammar.

Theorem 5.2 (Entropy of a Grammar): The Shannon entropy of a probabilistic context free grammar H(G) can be computed from

the entropies of the various random variables defining the grammar. Assume the base structure b_i is of the form $X_{i1}X_{i2} \ldots X_{iki}$ where each component is of the form L_j or D_j or S_j in the grammar. Then:

$$H(G) = H(B,R) = H(B) + H(R \mid B) \quad (5.2)$$
$$= H(B) + \sum_i [H(X_{i_1}) + H(X_{i_2}) + \cdots + H(X_{i_{k_i}})]$$

The proof of the above is fairly straightforward from the definitions of joint and conditional entropy. Using Theorem 5.2, for the simple grammar of Table 1 we have H(*G*) =3.42.

Note that we now have two different ways to calculate the Shannon entropy of our probabilistic distribution G: one is through generating the password guesses directly and computing the entropy and the other is using the grammar itself through Theorem 5.2. Clearly these should be the same. We did a few experiments to verify this on a few small sets of real user passwords and the entropy values came out as expected.

5.4 Increasing Shannon Entropy

We did a small experiment to test our updating approach. The experiment was done on a password training set of 740 real user passwords randomly chosen from the MySpace set which resulted in 37667 password guesses. We selected the new password for the user in such a way that its probability was less than or equal to 1/*n*, *n* being the total number of passwords in the distribution. Then the probability of the base structures and other components were updated with the technique described in the previous subsection. These steps were repeated until there was no password with probability less than 1/*n* (the distribution became uniform). The theoretical Shannon entropy value for this uniform distribution is 15.2. Figure 2 shows the changes in the Shannon entropy for each update round. As is evident, the system seems to be approaching the theoretical maximum Shannon entropy. We found a similar result for the guessing entropy.

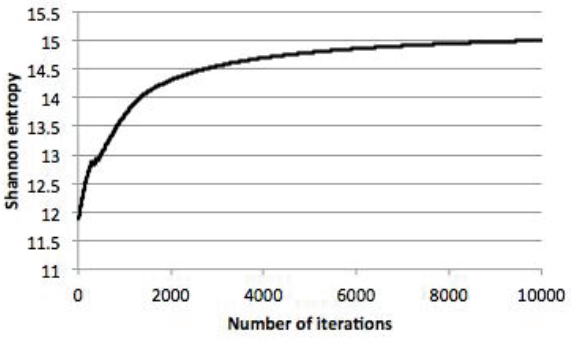


Figure 2. Shannon entropy changes with the update algorithm

Theoretically, having a uniform distribution for passwords is ideal since in that distribution all passwords will have equal probabilities. Practically, this would mean that each password is equivalent to being randomly chosen. Note that using our update algorithm we are moving closer to a uniform distribution but are likely very far away from it. For example, we only use words proposed by users and only modify the case. Thus, we will not use the full key space of alpha strings and it is similarly unlikely that we will ever exhaust the space of all 10 digit numbers. Nevertheless, while maintaining usability, we believe that our grammar modifying approach will ensure that an attacker cannot take advantage of using a probabilistic password cracking approach. Note that in our update algorithm, when updating the training set, we are not changing the probabilities of the passwords directly, but we are only changing the password distribution implicitly by changing the context-free grammar. Thus it is not obvious that we would approach the maximum Shannon entropy possible for that grammar.

6. TESTING THE AMP SYSTEM

6.1 Preprocessing and Experiment Setup

We tested the effectiveness of our analysis and modification system on several sets of revealed passwords. We call the AMP analysis component the *AMP password checker*. The grammar of this password checker is used to set the thresholds between strong and weak passwords and do the analysis of user proposed passwords. We used two different password cracking approaches to try to break passwords, including those that had been identified as weak and made strong by our AMP system.

We obtained three different lists of revealed passwords for our experiments. The first list is the RockYou password list [28], which was released in December 2009 and contains 32 million passwords. We used 2 million random plain text passwords from this list for our experiments. The second list is a MySpace password list, which contains 61,995 plain text passwords and was the result of an attack against MySpace users in 2006 [29]. The third list is the result of an attack against Hotmail users in October 2009 and contains 9,748 plain text passwords [30].

We randomly split each of these lists into separate sets for (1) training the AMP password checker (RockYou: 1 Million, MySpace: 30,997, Hotmail: 4874); (2) testing the AMP system (RockYou: ½ Million, MySpace: 15,499, Hotmail: 2,437); and (3) training a probabilistic password cracker (RockYou: ½ Million, MySpace: 15,499, Hotmail: 2,437). Note that the probabilistic password cracker is thus trained on a different set than is used for the AMP password checker. For the training sets, we combined passwords from the RockYou, MySpace, and Hotmail lists together since we wanted to have a comprehensive set for the training and because these different websites might have had different password policies for required lengths and other rule-based restrictions. We used "common_passwords" [4] and "dic-0294"[31] as input dictionaries to both our AMP checker and the probabilistic password cracker. Note that in our password checker we do not actually check the alphabetical part of the password against the dictionary; we assume the alphabetical part is included in the dictionary and we just use the probability value of words of that length for that component.

We set the threshold value for our experiments using the first approach in Section 3 and the AMP password checker grammar to generate the guesses and their probability values. The results were shown in Table 2. Note that the times shown in this table are the corresponding times for performing an MD5 hash on that number of guesses on the specific machine we used for cracking. At this point we have completed the preprocessing phase of the AMP system and can set a threshold as desired. Note that the Shannon entropy value for this grammar calculated by Theorem 5.2 is 26.78.

6.2 Implementation

The user interface of AMP (written in Java) takes either an individual password as input or a list of passwords. It checks the probability of the user proposed password against the threshold and tries to strengthen it within edit distance one if the password is weak. Note that we might not be able to strengthen some passwords since we currently only modify within edit distance of one in a certain way (such as keeping the alphabetical part). If a

password has a high probability and has only a small number of components we might not be able to alter it using an edit distance of 1 to get the probability below the threshold value. In our experiments we set the threshold value equivalent to different time periods; for example, one day (24 hours), meaning that a password is called weak if it can be cracked within one day, and it is strong otherwise. A one day threshold value is obviously not an ideal value in real life and we use other values in later tests. Figure 3 shows a snapshot of the AMP system with the user proposed password "life45!" as the input. The probability of the user-selected password as well as the probability value of the new password is shown along with the approximate time to crack.

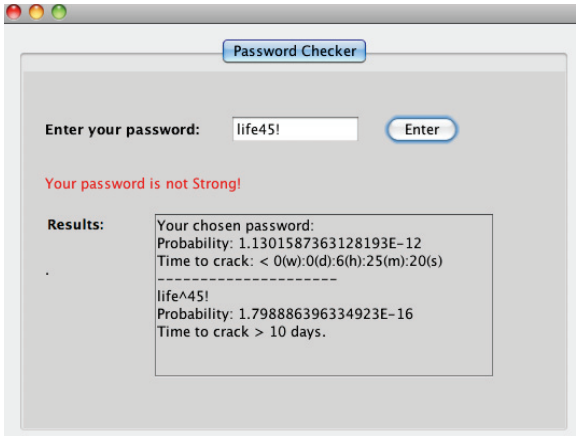


Figure 3. Snapshot of the AMP user interface

6.3 Password Cracking Results

To test our password analysis, we used two password cracking systems: (1) John the Ripper [4]; and (2) a probabilistic password cracker. We ran John the Ripper in incremental mode, which according to the documentation is their most powerful cracking mode and eventually tries all possible character combinations as passwords [4].

In the first series of results that we describe, the threshold value for the password checker was set to one day; thus we limited the number of guesses made by both password crackers to 43.2 billion guesses. This number is the approximate number of guesses that can be hashed by our test machine in one day.

In order for us to be able to compare our password checker with other existing password checkers (such as Microsoft password meter) we needed to be able to map the threshold value *thp* of our password checker to their score of weak and strong passwords. For example Microsoft outputs weak, medium, strong, and best as the result of password checking. Since we do not know their algorithm to score the strength of passwords, this comparison was not possible.

Table 5 and 6 show the results of the password cracking by John the Ripper and the Probabilistic Password Cracker (PPC) respectively, for a one day threshold. We divided the cracking results when using the AMP system into four different groups. We refer to the first group as *originally strong*. These are passwords that have been determined as strong based on the set threshold by AMP. The second group is called *originally weak not able to make stronger*, and are passwords that have been recognized as weak by AMP and the system has tried all possible modifications within distance one to strengthen them, but the modified probability values was not small enough to be below the threshold. The third group of passwords, referred to as *originally weak passwords able to make stronger,* are passwords that were determined as weak passwords by AMP and for which the AMP system was also able to strengthen them with modifications within edit distance one. Note that this set is the weak passwords without modification. The associated modified passwords are in the fourth group, which is consequently called *strengthened passwords modified from weak* in the tables. Note that these are now strong passwords are determined by the AMP system (relative to the threshold). Results show that both originally strong and strengthened passwords modified from weak passwords have very low rate of cracking compared with weak passwords. John the Ripper generally cracked less than 1% of the strong passwords and the Probabilistic Password Cracker cracked about 5%.

Table 5. Password cracking results using John the Ripper

	Originally **Strong** Passwords	Originally **Weak** Passwords (Not able to make stronger)	Originally **Weak** Passwords (Able to make stronger)	**Strengthened** Passwords Modified from Weak of previous column
Hotmail				
cracked / *total*	2 / 325	49 / 53	988 / 2,059	2 / 2,059
Percentage	(0.61%)	(92.45%)	(47.98%)	(0.0975%)
MySpace				
cracked / *total*	23 / 1484	104 / 149	5,343 / 13,866	71 / 13,866
Percentage	(1.55%)	(69.80%)	(38.53%)	(0.51%)
RockYou				
cracked / *total*	281 / 32,794	22,248 / 24,745	235,302 / 442,461	1,186 / 442,461
Percentage	(0.86%)	(89.90%)	(53.18%)	(0.27%)

Table 6. Password cracking results using PPC

	Originally **Strong** Passwords	Originally **Weak** Passwords (Not able to make stronger)	Originally **Weak** Passwords (Able to make stronger)	**Strengthened** Passwords Modified from Weak of previous column
Hotmail				
cracked / *total*	1 / 325	53 / 53	1,069 / 2,059	113 / 2,059
Percentage	(0.3%)	(100%)	(51.91%)	(5.48%)
MySpace				
cracked / *total*	27 / 1,484	135 / 149	8,341 / 13,866	698 / 13,866
Percentage	(1.81%)	(90.60%)	(60.15%)	(5.03%)
RockYou				
cracked / *total*	467 / 32,794	24,378 / 24,745	259,027 / 442,461	18,134 / 442,461
Percentage	(1.42%)	(98.51%)	(58.54%)	(4.1%)

Figure 4 shows how fast the weak passwords (that could be strengthened) were able to be cracked. With only 1 billion password guesses, which can be made in less than one hour, about 20% of MySpace passwords, 30% of Hotmail and 35% of RockYou passwords were cracked and in one day these numbers reached to 38%, 48% and 53% respectively.

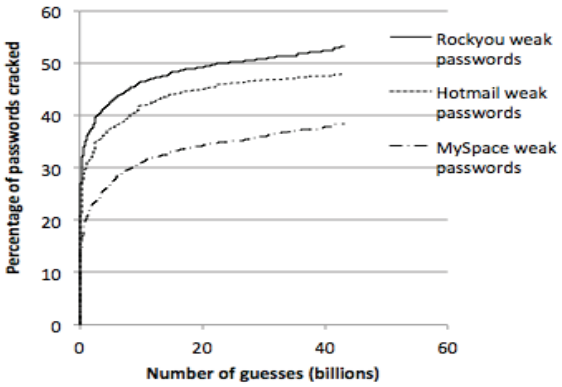


Figure 4. Weak passwords cracked using JTR

The analogous curve for strengthened passwords would be close to a straight line very close to the X-axis if overlaid in Figure 4 since less than 1% of them have been cracked in this period of time (24 hours). This curve is shown in Figure 5. This figure shows the percentage of strengthened passwords cracked over time by John the Ripper. (Note that the maximum percentage in this graph is 0.5% and not 50%.) These passwords are the strengthened version of the passwords from the previous figure that have been made strong by the AMP system. These two figures show how much the rate of cracking has been decreased after modification.

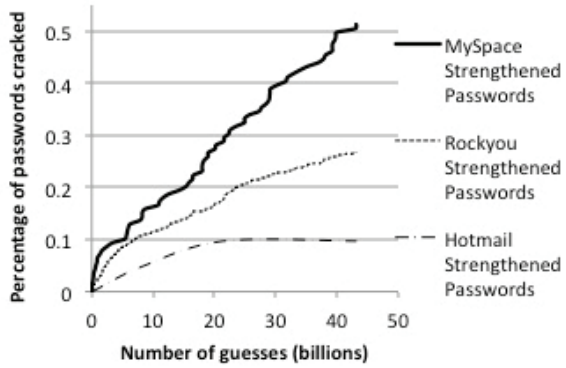


Figure 5. Strengthened passwords cracked using JTR

Overall, before using AMP, the total rate of cracking the test password set (columns 1,2,3) was 56.6% with the probabilistic password cracker. After using AMP, since it only allows strong passwords (columns 1 and 4) the cracking rate is 3.9%. Also note that the analysis system successfully determines weak passwords from strong ones with an error rate of 1.43% (column 1). This rate is the percent of passwords originally identified as strong, but that can be cracked. Notice that we do identify a fair number (39%) of passwords as weak but that we were unable to crack (columns 2 and 3). We believe that this type of error is acceptable as it only means that we are being conservative and we could not crack some passwords that we deemed weak.

Besides using the 1 day threshold, we also ran similar tests as the above using threshold values (see Table 2) for 12 hours, 48 hours and 96 hours. Figure 6 shows the total rate of cracking the test password set before using AMP and after using AMP for both John the Ripper (JTR) and the probabilistic cracker (PPC). The time allocated for cracking was of course the same time as used for determining the threshold. Note that the results are similar to the 1 day results and even at 4 days we are significantly improving the weak passwords.

We could not strengthen some of the weak passwords since we are limiting changes to edit distance one. Furthermore, we did not have any restrictions on the proposed passwords such as minimum length. The identified weak passwords that we could not strengthen were 4.0%, 4.8%, 18.6% and 37.7% (of the total test password set) for 12 hours, 24 hours, 48 hours, and 96 hours, respectively. As an example, we could not strengthen the password "123456" in our data set. This is to be expected and could likely be remedied by allowing edit distance two or having some minimal restrictions on the input password.

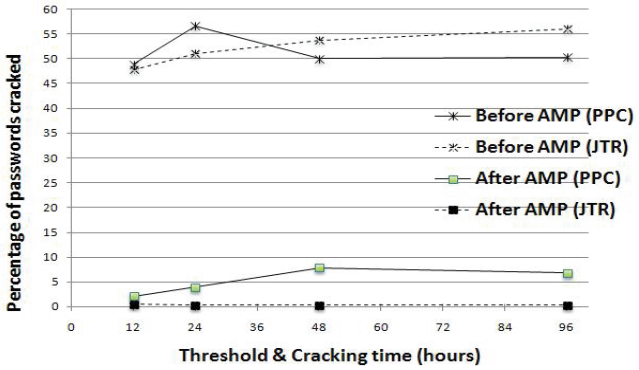


Figure 6. Using AMP for different time thresholds

7. CONCLUSION AND FUTURE WORK

We developed a new approach to help users create strong passwords based on determining a threshold value that rejects those that are weak and then slightly modifies the passwords to make them strong. We used a probabilistic context-free grammar to estimate the probability of a password being cracked. This grammar is derived through training on large sets of revealed passwords. The grammar can be used to generate the passwords in highest probability order so that it is easy to calculate Shannon entropy and guessing entropy. We also discussed a way to calculate the Shannon entropy directly from the grammar. The reject function was based on using a threshold probability that could be determined in several ways and separated weak from strong passwords. We built a system (AMP) and showed the effectiveness of our approach through a series of experiments. We further showed how to use a dynamic update algorithm to change the grammar as the system generated passwords to preserve the ability of the system to generate strong passwords. We also showed that the update algorithm would drive the grammar to higher Shannon entropy and thus the new password distribution (including passwords generated by AMP) would be increasingly resistant to password cracking.

In future work, we want to explore fast implementations of finding passwords with edit distance greater than one. We also want to further explore the modification algorithm to see what operations are most effective in generating usable passwords and investigate the security of the system if all aspects of the algorithms and thresholds were to be known. By knowing the threshold, the attacker could possibly eliminate passwords above threshold in our approach. Clearly this also applies to other techniques, for example rule based policies. It is obvious if the rules were known for scoring a password that those passwords that do not pass the rules can be ignored in attacks. Note that in our approach however, the distributions of passwords (with respect to probability) always tend to flatten out and thus it

becomes harder to crack passwords as the threshold is lower. In our work we randomly generate changes to a user provided password, but this original password is *not known* to the attacker. Thus simply guessing all the variations on passwords based on our modifying algorithm is problematic.

We intend to explore these issues in future work but we believe that our approach is still better than existing approaches with respect to security of the algorithms. Note that our dynamic update algorithm constantly changes the probabilities of the grammar after any password is strengthened and moves towards a more uniform probability distribution. This would be difficult to model unless the attacker also had all the proposed passwords that we modified.

8. REFERENCES

[1] M. Weir, S. Aggarwal, M. Collins, and H. Stern, "Testing metrics for password creation policies by attacking large sets of revealed passwords," Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS '10), October 4-8, 2010, pp. 163-175.

[2] E. R. Verheul, "Selecting secure passwords," M. Abe (Ed.): CT-RSA 2007, LNCS 4377, pp 49-66, 2007.

[3] M. Weir, Sudhir Aggarwal, Breno de Medeiros, Bill Glodek, "Password Cracking Using Probabilistic Context Free Grammars," Proceedings of the 30th IEEE Symposium on Security and Privacy, May 2009, pp. 391-405.

[4] The Open wall group, John the Ripper password cracker, http://www.openwall.com.

[5] Shannon Riley, "Password security: what users know and what they actually do," Usability News, 8(1), 2006.

[6] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. Kemmerer, C. Kruegel, and G. Vigna, "Your botnet is my botnet: Analysis of a botnet takeover," Tech. Rep., April 2009.

[7] R. Shay, S. Komanduri, P. G. Kelley, P. G. Leon, M. L. Mazurek, L. Bauer, N. Christin, and L. F. Cranor, "Encountering stronger password requirements: user attitudes and behaviors," In 6th Symposium on Usable Privacy and Security, July 2010.

[8] Furnell, S., "An assessment of website password practices," Computers & Security 26, 7-8 (2007), 445-451.

[9] W. Burr, D. Dodson, R. Perlner, W. Polk, S. Gupta, E. Nabbus, "NIST special publication 800-63-1 electronic authentication guideline," National Institute of Standards and Technology, Gaithersburg, MD, April, 2006.

[10] Y. Zhang, F. Monrose, and M. K. Reiter, "The security of modern password expiration: an algorithmic framework and empirical analysis," In Proceedings of ACM CCS'10, 2010.

[11] Philip G. Inglesant, M. Angela Sasse, "The true cost of unusable password policies: password use in the wild," Proc. of the 28th international conference on Human factors in computing systems, April 10-15, 2010, Atlanta, Georgia.

[12] Charoen, D., Raman, M., and Olfman, L., "Improving end user behavior in password utilization," Systemic Practice and Action Research, 21(1), 55. 2008.

[13] A. Adams and M. A. Sasse, "Users are not the enemy," Communications of the. ACM, 42(12):40–46, 1999.

[14] J. Campbell, W. Ma, D. Kleeman,"Impact of restrictive composition policy on user password choices," Behavior and information technology, Vol. 30, No. 3, May-June 2011.

[15] Florencio, D. and Herley, C., "A large-scale study of web password habits," In Proceeding of the 16th Int. Conf. on World Wide Web, 2007.

[16] R. Shay, S. Komanduri, P. G. Kelley, P. G. Leon, M. L. Mazurek, L. Bauer, N. Christin, L. F. Cranor, and S. Egelman, "Of passwords and people: measuring the effect of password-composition policies," Proceeding of 2011 Annual Conference on Human Factors in Computing Systems, 2011.

[17] G. Bard, "Spelling-error tolerant, order independent pass-phrases via the Damerau-Levenshtein string-edit distance metric," Fifth Australasian Symposium on ACSW Frontiers - Volume 68 (Ballarat, Australia, January 30 - February 02, 2007), 117-124.

[18] Yan, J. J., Blackwell, A., Anderson, R. and Grant A., "The memorability and security of passwords -- some empirical results," Technical Report No. 500 (September 2000) Computer Laboratory, University of Cambridge.

[19] A. Forget, S. Chiasson, P.C. van Oorschot, R. Biddle, "Improving text passwords through persuasion," Symposium on Usable Privacy and Security (SOUPS) 2008, July 23–25, 2008, Pittsburgh, PA USA.

[20] J. Yan, "A note on proactive password checking," ACM New Security Paradigms Workshop, New Mexico, USA, 2001.

[21] EH Spafford, "OPUS: preventing weak password choices," Computers& Security (1992).

[22] S. Schechter, C. Herley, M. Mitzenmacher, "Popularity is everything: a new approach to protecting passwords from statistical-guessing attacks", HotSec'10: Proceedings of the 5th USENIX conference on Hot Topics in Security, 2010.

[23] C. Castelluccia, M. Durmuth, D. Perito, "Adaptive password-strength meters from Markov models," NDSS '12, 2012.

[24] C. E. Shannon, "Prediction and entropy of printed English," Bell Systems Tech. J., vol. 30, pp 50-64, Jan. 1951.

[25] J. Massey, "Guessing and entropy," 1994 IEEE Symposium on Information Theory, pp. 204, 1994.

[26] Damerau, F. J. "A technique for computer detection and correction of spelling errors. Communications of the ACM, vol. 7, Issue 3, pp. 171-176, March 1964.

[27] T. M. Cover and J. A. Thomas, Elements of Information Theory, Wiley, 1991.

[28] A. Vance, "If your password is 123456, just make it hackme," New York Times, January 2010, http://www.nytimes.com/2010/01/21/technology/21password.html.

[29] Robert McMillan, "Phishing attack targets MySpace users," http://www.infoworld.com/d/security-central/phishing-attack-targets-myspace-users-614, October 27, 2006.

[30] T. Warren, "Thousands of Hotmail Passwords Leaked," http://www.neowin.net/news/main/09/10/05/thousands-of-hotmail-passwords-leaked-online.

[31] A list of popular password cracking wordlists, 2005, http://www.outpost9.com/files/WordLists.html.

Cloud-Based Push-Styled Mobile Botnets: A Case Study of Exploiting the Cloud to Device Messaging Service

Shuang Zhao[1,2], Patrick P. C. Lee[3], John C. S. Lui[3], Xiaohong Guan[1], Xiaobo Ma[1], Jing Tao[1]
[1]School of Electronic & Information Eng., Xi'an Jiatong University, China
[2]Institute of Information Engineering, Chinese Academy of Sciences, China
[3]Dept of Computer Science & Engineering, The Chinese University of Hong Kong, Hong Kong
dflower.zs@gmail.com, {pclee,cslui}@cse.cuhk.edu.hk,
xhguan@xjtu.edu.cn, superxiaoboma@gmail.com, jtao@xjtu.edu.cn

ABSTRACT

Given the popularity of smartphones and mobile devices, mobile botnets are becoming an emerging threat to users and network operators. We propose a *new* form of cloud-based push-styled mobile botnets that exploits today's push notification services as a means of command dissemination. To motivate its practicality, we present a new command and control (C&C) channel using Google's Cloud to Device Messaging (C2DM) service, and develop a *C2DM botnet* specifically for the Android platform. We present strategies to enhance its *scalability* to large botnet coverage and its *resilience* against service disruption. We prototype a C2DM botnet, and perform evaluation to show that the C2DM botnet is *stealthy* in generating heartbeat and command traffic, *resource-efficient* in bandwidth and power consumptions, and *controllable* in quickly delivering a command to all bots. We also discuss how one may deploy a C2DM botnet, and demonstrate its feasibility in launching an SMS-Spam-and-Click attack. Lastly, we discuss how to generalize the design to other platforms, such as iOS or Window-based systems, and recommend possible defense methods. Given the wide adoption of push notification services, we believe that this type of mobile botnets requires special attention from our community.

1. INTRODUCTION

With the advent of mobile Internet access, we have seen significant technological advancement of smartphones and mobile devices. This provides a fertile ground for hackers to realize *botnets*[1] in a mobile network. In recent years, we have seen many real-life examples of mobile botnets. In 2009, a mobile botnet SymbOS.Yxes [1] was discovered in the Symbian platform, which used the conventional HTTP-based C&C channel for communication. In December 2010, the first iPhone botnet Ikee.B [2] was found in the wild. It targeted jailbroken iPhones and pulled commands from a HTTP server. In the same year, the first Android botnet named GEINIMI [3] emerged. It also used a HTTP-based C&C channel for command dissemination. In 2011, a Short Messaging Service (SMS)-based mobile botnet named ZeuS [4] was found in the Blackberry, Symbian, and Windows Mobile platforms. It used an SMS-based C&C channel to communicate with the botmaster. In September 2011, an Android bot called *AnserverBot* was identified and it was the first Android bot that used public blogs as C&C servers [5]. In late January 2012, an Android botnet disguised as the game application "Madden NFL 12" [6] and used the Internet Relay Chat (IRC) as its C&C channel. All these incidents indicate that mobile botnets have become an emerging threat for users and network operators.

From a botmaster's perspective, how to deploy a stealthy and robust mobile botnet is an interesting issue; from an operator's perspective, understanding the deployment strategy of a mobile botnet is critical for defending against malicious attacks on an operational network. Our key observation is that many smartphone platforms provide the *push notification service*, which comprises a cloud of push-based messaging servers that are responsible for relaying messages from application servers to mobile applications. There are various deployments of this service in different platforms, such as Google's Cloud to Device Messaging (C2DM) service for Android [7], Apple's Push Notification Service (APNS) [8] for iOS, Microsoft's Push Notification Service (MPNS) for Windows Mobile [9], Blackberry's Push Service (BPS) [10], and Nokia's Notifications API (NNA) for Symbian and Meego devices [11]. Architectures of these push notification services have one common feature: application servers first send a notification message with an intended receiver (or the target mobile device) to one of the cloud-based messaging servers, which then *pushes* the message to the target mobile device. The push notification service eliminates the needs of application servers to keep track of the state of a mobile device (i.e., active or offline). Furthermore, mobile devices do not need to periodically probe the application servers for messages, thereby reducing the workloads of the application servers. While the push notification service simplifies the mobile application development, it can also be exploited by attackers in building a *highly potent* and *stealthy* mobile botnet compared to traditional HTTP, IRC, or SMS botnets.

This paper aims to expose such kind of potential attack scenario. As a proof of concept, we consider Google's C2DM service for the Android platform, and realize a cloud-based push-styled mobile botnet using the push notification service as a C&C channel. We named it as *C2DM botnet*[2], which involves no direct communication between the botmaster and various bots, but instead ex-

[1]A *botnet* is a network of compromised computers called *bots* that are remotely controlled by a *botmaster*.

[2]We note that Google's C2DM service is deprecated on June 26, 2012, after the paper is submitted for review. Nevertheless, the fundamental design of the C2DM botnet remains applicable to the next-generation C2DM service. We discuss this issue in §7.

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

ploits Google's C2DM service as a relay. The botmaster can disseminate probes and commands to the bots via the C2DM service, and the botnet traffic can be "hidden" within the C2DM traffic of other legitimate mobile applications. This makes the C2DM botnet stealthy. In summary, we make several contributions in motivating the practicality of this new form of mobile botnets:

• **Design and implementation of a cloud-based push-styled mobile botnet.** We build a C2DM botnet for the Android platform using Google's C2DM push notification service as the C&C channel. We propose a multiple-username strategy to enhance its *scalability* to a large botnet size, and propose how to make the botnet *resilient* against service disruption due to account unregistration.

• **Performance evaluation of the C2DM botnet.** We extensively evaluate the C2DM botnet design, and show its (i) *stealthiness* in generating heartbeat and command traffic by hiding its existence under other legitimate C2DM traffic, (ii) *resource efficiency* in bandwidth and power consumptions compared to traditional IRC and HTTP bots, and (iii) *controllability* in disseminating commands to all bots within a short period of time.

• **Demonstration of C2DM bot propagation and attack.** We show how hackers can infect and propagate a C2DM bot using legitimate mobile applications, and show how this botnet can launch an effective SMS-Spam-and-Click attack.

• **Extension to other mobile platforms.** We discuss how the C2DM botnet can be extended to other platforms, such as iOS and Microsoft Windows.

• **Recommendation of potential defense strategies.** We suggest detection and defense strategies against this underlying threat.

The rest of the paper proceeds as follows. §2 reviews related work. §3 presents an overview for the C2DM service. §4 presents the design and implementation of a C2DM botnet along with strategies that enhance its scalability and fault tolerance. §5 presents evaluation results for the C2DM botnet. §6 discusses how an attacker may propagate a C2DM bot and feasibly launch an attack. §7 shows how to generalize the C2DM botnet design to other platforms. §8 discusses potential defense strategies, and §9 concludes.

2. RELATED WORK

There are a number of studies in the literature on mobile botnets. Traynor et al. [12] study how a mobile botnet launches a DDoS attack against a cellular network core. They show that a small-size mobile botnet is sufficient to cause nationwide outages. Singh et al. [13] propose a mobile botnet using Bluetooth as a C&C channel. The commands are disseminated from one bot to another within the radio range of bluetooth. Their experiments show that commands from the botmaster can reach about 2/3 of bots in 24 hours. Xiang et al. [14] discuss the single point of failure problem in traditional centralized HTTP botnets. They propose a mobile botnet named URL Flux to make a traditional HTTP-based botnet more resilient against existing defense solutions.

Short Message Service (SMS) can also be used as a C&C medium for mobile botnets. Zeng et al. [15] use SMS messages for C&C to build mobile botnets. They leverage on the P2P topological structure in the botnet to reduce the number of SMS messages being sent and shorten the time delay for delivering SMS commands. Each bot needs to send SMS messages to its neighbors in order to join the P2P network and forward commands. Geng et al. [16] propose an SMS-based mobile botnet, in which they divide bots into normal bots and regional bot servers to build a P2P network in order to reduce the forward and search consumptions, and enhance the robustness of the botnet. Hua et al. [17] design a SMS-based mobile botnet and evaluate its construction under different topologies. Their simulations show that in a botnet containing 20,000 bots, a command from the botmaster could be covertly disseminated to over 90% of all bots, and each bot only needs to send no more than four SMS messages during the dissemination. Weidman [18] design a transparent C&C channel for a SMS-based botnet, in which the bot can intercept and read commands from incoming SMS messages before the messages are presented to users. Mulliner and Seifert [19] combine the Kademlia P2P network with an SMS-HTTP hybrid approach as the C&C channel for a mobile botnet, in which the communication is split into HTTP and SMS parts. We want to emphasize that one disadvantage of SMS-based botnets is in the cost incurred when sending commands via SMS, especially if the botnet is of large scale. Also, the use of SMS may eventually alert users and mobile operators since sending/receiving SMS messages may incur payment to mobile phone operators. In our proposed botnet, bots communicate with the C&C servers via the Internet, and this is much cheaper and stealthier.

Several botnet detection methods (e.g., [20, 21, 22, 23]) used in traditional wireline networks are also applicable to detect centralized IRC- and HTTP-based mobile botnets. Specifically for mobile botnets, Vural et al. [24] use a forensics-based approach: they first model normal activities of mobile users, and use the model results to identify the abnormal activities of malware or botnets.

3. OVERVIEW OF C2DM

We now describe Google's cloud-based push-styled messaging service, or the *Cloud to Device Messaging* (C2DM) service (as illustrated in Figure 1). It enables third-party developers to send push notifications to their mobile applications on Android devices.

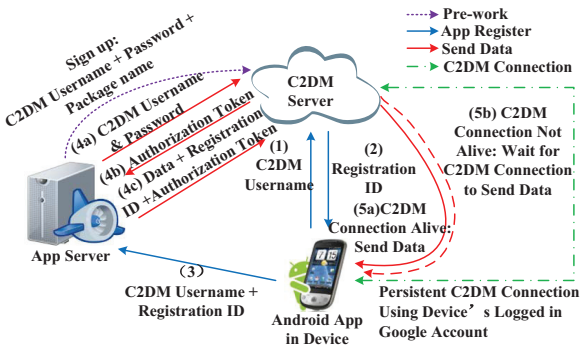


Figure 1: C2DM architecture and its workflow.

To bootstrap the C2DM service, the application developer has to first sign up for an account. This is done by providing the C2DM server with the following: *C2DM username*, *password*, and the *package name* of the mobile application. Once the account is established, the developer can embed the C2DM username in the mobile application, and distribute the application, say, via the official Android Market or other third-party marketplaces.

We now elaborate the workflow of the C2DM service as depicted in Figure 1. When the mobile application is first launched in a mobile device, it will perform the following steps.

• **Step (1):** The mobile application registers itself to one of the C2DM servers using the C2DM username, which was provided by the application developer, as well as the *device ID*, which uniquely identifies the Android device that hosts the application.

• **Step (2):** The C2DM server provides a unique *registration ID* to the mobile application. The registration ID is a byte string that enables the C2DM server to identify the application running on a specific Android device.

• **Step (3):** The mobile application sends this registration ID, together with its C2DM username, to the application server, which will then record this registration ID in its database.
• **Step (4):** When the application server needs to send data to a mobile device, it sends the C2DM username and password to the C2DM server (Step (4a)), and gets an authorization token if the username and password are valid (Step (4b)). The authorization token will be used to notify a set of mobile devices in the database. The application server then sends a C2DM request, which contains the notification message, the registration ID of a mobile application, and the authorization token to the C2DM server (Step (4c)). Note that the message in Step (4c) is sent on a *per-device* basis. Thus, if there are k devices that need the notification, then the application server will send k messages to the C2DM server.
• **Step (5):** Upon receiving the message, the C2DM server looks for the specific Android device based on the registration ID. If the C2DM connection of that device is alive, then the C2DM server will send the notification message to the mobile application on that device (Step (5a)); if the mobile device is disconnected, then the C2DM server will store the notification message, and send the message to the application on the mobile device when the device re-establishes its connection with the C2DM server (Step (5b)). Note that Step (5) implies that there is a *persistent C2DM connection* between the C2DM server and the Android device that subscribes to the C2DM service.

Google's C2DM service provides a flexible solution for developers to send lightweight messages to mobile applications without requiring mobile devices to connect to their servers periodically to pull for messages. Google manages the storage and forwarding of messages. Thus, it simplifies the mobile application design, and reduces the network traffic and battery usage of mobile devices. C2DM maintains a persistent TCP connection for each mobile device with a C2DM server, and the time interval for each heartbeat message of this persistent connection is around 15-30 minutes depending on the device state (i.e., ACTIVIT, IDLE, SYNC, NOSYNC). When a new C2DM message arrives, the mobile device will wake up the mobile application to receive the message.

The C2DM service is available for a device running on Android 2.2 or higher versions. The device must have the Market (now Google changes its name to Play Market) application installed and at least one logged-in Google account [7]. Market is one of the factory-installed applications in the Android platform, along with other *popular* mobile applications like Google Maps, Gmail, etc. Users are required to log in with a Google account and enable the C2DM service when using these applications. There are also many other popular applications which rely on the C2DM service, such as Instagram, Facebook for Android, and LINE. Thus, we expect that the C2DM service is enabled in majority of Android devices.

4. DESIGN OF A C2DM BOTNET

This section presents the design and implementation of a C2DM botnet. Our design is based on the official and open C2DM architecture. To ease our presentation, we first discuss a baseline architecture for a C2DM botnet. Then we present an enhanced architecture that has a stronger stealthiness property. Finally, we address the scalability and fault-tolerance issues.

4.1 Baseline C2DM Botnet Architecture

We first describe the baseline architecture of a C2DM botnet, as shown in Figure 2. Before building a C2DM botnet, the botmaster first bootstraps a C2DM service as for other normal C2DM applications (see §3). Then the steps of how a new bot joins the C2DM botnet and how the botmaster disseminates commands can be done as follows (referring to Figure 2).

• **Bot registration** (Steps (1) to (3)): When a new bot joins a C2DM botnet, it first registers itself to one of the Google C2DM servers with the following information: (i) the package name and the C2DM username, both of which are embedded by the botmaster in the distributed malware package, and (ii) the Google's account ID of the mobile device. If the registration is successful, then the C2DM server will return a unique registration ID to the bot. Finally, the bot sends the registration ID and the C2DM username to the botmaster's C&C server, which will then record this registration ID in its database for future command dissemination.
• **Command dissemination** (Steps (a) to (c)): The botmaster disseminates commands to all registered mobile bots via a C&C server. The C&C server first authenticates itself to the C2DM server by using its Google's account ID and password. After authentication, the C2DM server will return an authorization token to the C&C server, which then constructs a C2DM request for *each* mobile device (or bot). The request contains the command by the botmaster, the registration ID of the bot, and the authorization token. The C&C server then sends the request to the C2DM server. Based on the registration ID, the C2DM server will push the message to the bot. Note that the *C2DM service only allows one registration ID in each C2DM request*. This implies that sending commands to multiple bots requires the C&C server to send multiple requests. We will discuss how to scale up a botnet and control the message dissemination in §4.3, and how to control message delivery in §5.3.

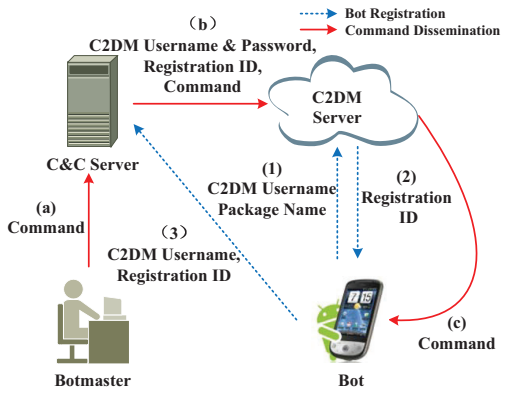


Figure 2: Baseline architecture of a C2DM botnet.

4.2 Enhanced C2DM Botnet Architecture

In the baseline architecture, each bot needs to directly send its registration ID to the C&C server. This increases the possibility of being detected and reveal the identity of the C&C server. To improve the stealthiness, the enhanced architecture eliminates the direct communication between a bot and its botmaster.

Before building a C2DM botnet, the botmaster needs to sign up *two* C2DM accounts: (i) C2DM username_M, which is used by the botmaster's C&C server to send messages to its bots, and (ii) C2DM username_B, which is used by the bots to send messages to the C&C server. The botmaster needs to register itself to a C2DM server using C2DM username_B to obtain a registration ID, which will later be used by the bots to send C2DM messages to the C&C server. We assume that C2DM username_M, C2DM username_B, and the botmaster's registration IDs are all embedded in the malware package.

Figure 3 depicts the enhanced architecture. The steps of bot registration and command dissemination are revised as follows.

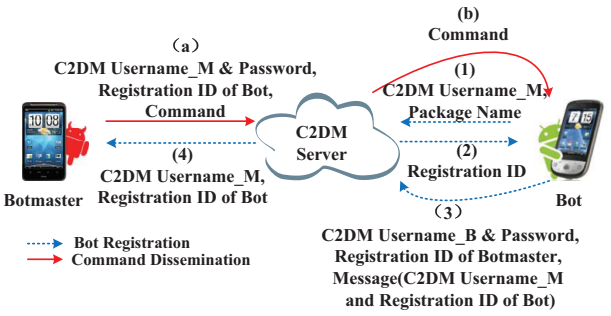


Figure 3: Enhanced architecture of a C2DM botnet.

• **Bot registration** (Steps (1) to (4)). When a new bot joins a C2DM botnet, it registers itself to one of the C2DM servers using its package name and C2DM username_M. If the registration is successful, then the C2DM server will return a unique registration ID to the bot. Then the bot uses the account of C2DM username_B and the botmaster's registration ID (both of which are embedded in the malware package) to send its own registration ID and C2DM username_M to the botmaster, which records the information for later command dissemination.

• **Command dissemination** (Steps (a) to (b)). To disseminate commands to all registered mobile bots, the C&C server constructs a C2DM request for *each* bot. Unlike the baseline architecture, this request now contains the account C2DM username_M, the registration ID of the bot, and the command. The C&C server sends the request to the C2DM server, which will then be pushed to the corresponding bot based on the registration ID.

Remark on stealthiness and reliability. One major advantage of the enhanced architecture is that during the bot registration period, the bot sends its registration ID and C2DM username to the C&C server via the C2DM server. Thus, it is stealthier than the baseline architecture. However, one shortcoming of the enhanced architecture is that it relies on the botmaster's registration ID for communication, and Google may revoke any registration ID that is maliciously used. Although our experience is that Google seldom explicitly un-registers registration IDs, there is no guarantee that a registration ID will remain valid permanently. If the registration ID is revoked, then the bots cannot send messages to the botmaster. We will discuss how to overcome this issue in §4.4.

4.3 Scaling up the C2DM Botnet

As mentioned in §4.1, each C2DM request can only be sent to a single device. Also, some quota limits may be posed on the rate of push messages that can be delivered. For example, Google's C2DM service limits each account to send no more than 200,000 push messages per day [7]. Furthermore, when signing up for a new C2DM service, one needs to specify the estimated peak queries per second (QPS), which is a short-term push rate permitted for an application. In C2DM, there are four choices for QPS: "0 - 5", "6 - 10", "11-100" and "> 100". To ensure that the C2DM botnet is stealthy, we assume that the QPS of a C2DM botnet is less than 100 to avoid drawing Google's attention.

For a small-scale C2DM botnet, 200,000 push messages per account per day should suffice to disseminate commands to all the bots, and it takes only a short duration to disseminate commands to the whole botnet under our QPS requirement. We will discuss the controllability and estimate the time needed to disseminate commands to most of the bots in §5.3. However, using one C2DM account to maintain a botnet becomes problematic if (i) the botnet is of large size, or (ii) the botnet needs to send more than 200,000 push messages per day.

To build a large-scale C2DM botnet, we propose to use *multiple C2DM usernames* to decompose a large botnet into several smaller subnets. Each of these subnets uses a unique C2DM username to communicate with its own bots. We elaborate how to incorporate multiple C2DM usernames into a C2DM botnet, using the enhanced architecture as an example, as shown in Figure 4. Here, a bot joins the botnet using a *two-phase* C2DM registration process. The bot first registers to a C2DM server using an initial username, and then sends its own registration ID to the botmaster via the C2DM service (Step (1)). The botmaster, upon receiving the C2DM message, chooses one of the pre-defined C2DM usernames and sends it to the bot via C2DM (Steps (2) to (4)). Then the bot re-registers to a C2DM server with the new C2DM username and sends the new registration ID to the botmaster via C2DM (Steps (5) to (6)). The botmaster will then use the new registration ID to send commands to the bot.

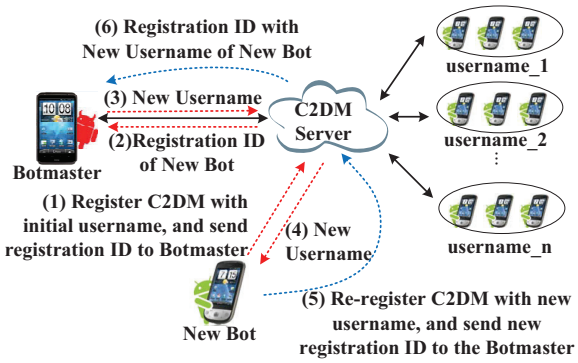


Figure 4: Registration in a large-scale enhanced C2DM botnet.

4.4 Handling Account Un-registration

Each botnet must deal with the single-point-of-failure problem. We now discuss how a C2DM botnet can self-configure in case a core component fails. In a C2DM botnet, we argue that the C&C server, though important, is not the core factor to consider due to two reasons. First, bots communicate with the C&C server only once during their registration. Once a botnet is built, bots will not directly communicate with the C&C server, so it is quite stealthy. Second, one may use well-known social networking websites such as Twitter, Facebook to set up a C&C server to enhance the stealthiness [14, 25].

On the other hand, we need to ensure the robustness of the C2DM service in a C2DM botnet. If the C2DM service is unavailable, then the communication between the botmaster and bots will be cut off. One way this may happen is that the C2DM service used by the bot is *banned* and *un-registered* by Google. When a bot registers to a C2DM server, it is required to provide both its package name and C2DM username. In our experiments, we find that when we sign up for a C2DM service, the package name need not be unique. In other words, one can sign up for the C2DM service using the same package name as other existing applications. This implies that Google is not likely to ban specific package names from the C2DM service, as it may unexpectedly ban other legitimate services. On the other hand, the C2DM username may be banned from any C2DM service, and this can shut down the communication of the entire C2DM botnet.

To overcome service disruption due to the un-registration of the C2DM username, one can again leverage the multiple-username

strategy proposed in §4.3. The main idea of such a strategy is to set up several backup C&C servers, and if a bot has not received any messages from the botmaster for a pre-defined duration (e.g., one week), then it will communicate with one of the backup C&C servers and query whether it needs to change its C2DM username. If it gets a new username, it will use the new username to register to the C2DM service again and sends the new registration ID to the C&C server in order to re-join the botnet. We point out that the backup C&C server also possesses the stealthiness property because bots seldom communicate with it.

5. EVALUATION

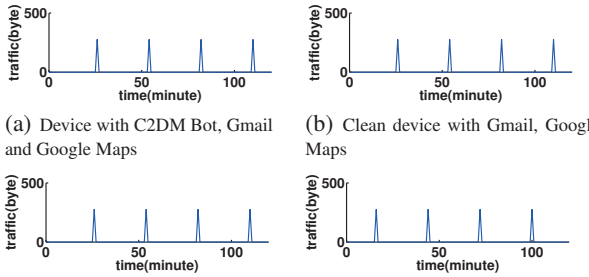
In the previous section, we showed that the C2DM botnet communication protocol possesses the stealthiness property: there is no *direct* communication between the botmaster (or C&C server) and bots. In fact, all communication and command dissemination are carried out via the C2DM service. In this section, we take a closer examination on the stealthiness property by measuring (a) the heartbeat overhead of maintaining a C2DM persistent connection; (b) the capability of hiding command dissemination among legitimate C2DM traffic, and (c) resource (i.e., bandwidth and power) consumption on mobile devices. We also explore the *controllability* of a C2DM botnet: if a C&C server wants to disseminate a command, then what is the *minimum time* needed for the botmaster to claim, with a high probability, that all bots receive the command?

5.1 Stealthiness in Control/Data Plane

We first examine the stealthiness of a C2DM botnet in terms of network traffic. System operators can perform online/offline network traffic analysis to detect a botnet [22, 23], and examine the following *suspicious behaviors*: (i) connecting to some unauthorized servers (i.e., C&C servers) using domain names, URLs, or IP addresses, and (ii) communicating with one or more servers with certain periodic patterns or with an abnormally high frequency. For example, a traditional HTTP bot will connect to a C&C server with an unauthorized domain name or IP address, and periodically pull commands from the C&C server. These periodic connections can expose the presence of a bot. An advanced HTTP bot may use URL flux [14], which uses authorized domain names such as "twitter.com" to pull commands, but frequent connections can also be classified as abnormal behavior.

We expect that a C2DM botnet has high stealthiness with the following intuition. In the baseline architecture (see §4.1), a bot connects to a C&C server only *once*, i.e., when sending its registration ID. Then during the keep-alive period, the bot will never communicate with the C&C server. In the enhanced architecture (see §4.2), a bot connects to a C&C server only when the registration ID is unregistered (see §4.4). Thus, the probability for a C2DM botnet to expose a bot or the C&C server is kept at minimum. In the following, we examine the stealthiness of a C2DM botnet by measuring the network traffic in both *control and data planes*. We implement a C2DM bot in the Android emulator [26], while the measurement results are also applicable for real Android phone devices.

Stealthiness of heartbeat traffic. We first look into the control plane, and focus on the periodic heartbeat messages. Note that if an Android phone enables any legitimate C2DM service, then it will connect to a C2DM server with a persistent TCP connection and sends periodic heartbeat messages to a C2DM server to check for any new push message. We compare the C2DM heartbeat traffic of two Android phones, one being installed with a C2DM bot and another being a clean Android phone. Both Android phones are installed with a number of legitimate applications that require the C2DM service. Figure 5 shows the traffic patterns of heartbeat messages under different settings, where the x-axis is the time (with unit in minutes) and the y-axis is the traffic volume (with unit in bytes). Figures 5(a) and 5(c) show the heartbeat traffic patterns for a phone with a C2DM bot, while Figures 5(b) and 5(d) show the heartbeat traffic patterns for clean a mobile phone, with three or five legitimate C2DM applications, respectively. We observe that the traffic patterns of these settings are *almost identical*, since all legitimate mobile applications which use the C2DM service share the same persistent TCP connection per device. This allows a C2DM bot to hide itself under legitimate C2DM heartbeat traffic. Since a C2DM bot relies on the existing C2DM service of the mobile phone, if the C2DM service is not enabled, then the bot will be dormant and will not generate any C2DM heartbeat traffic.



(a) Device with C2DM Bot, Gmail and Google Maps

(b) Clean device with Gmail, Google Maps

(c) Device with C2DM Bot, Gmail, Google Maps, Facebook, LINE and Instagram

(d) Clean device with Gmail, Google Maps, Facebook, LINE and Instagram

Figure 5: C2DM traffic: (a) and (c) are traffic for a C2DM bot; (b) and (d) are traffic for clean devices, with 3 or 5 apps.

Stealthiness of command dissemination. We examine the stealthiness of a C2DM botnet in the data plane. Note that many Android phones typically come with a number of *pre-installed* applications such as Gmail, Google Maps, Google Play, or some *popular* applications like Facebook, LINE, Whatsapp, etc. All these applications use C2DM services. Upon installation, the C2DM bot knows which of these legitimate C2DM applications are installed in the mobile phone, and this provides valuable information to enhance its stealthiness. For example, in [27], authors indicate the interarrival time of Gmail messages is about 0.53 hours. Thus, if a botmaster disseminates commands with an average interarrival time longer than those of the legitimate C2DM applications, then it is easy to hide the existence of a C2DM bot.

We perform experiments by installing a C2DM bot into an Android phone that is pre-installed with Gmail and Google Maps. We then measure the data traffic generated by the bot and the legitimate applications Gmail and Google Maps as follows. During the experiment, the C&C server sends commands to the bot. In the same measurement period, the phone also receives several email notifications from Gmail and check-in notifications from Google Maps. Figure 6 shows the data traffic generated by the bot and the legitimate applications in one particular measurement. We observe that the C2DM botnet traffic only occupies less than 20% of the overall data traffic. The size of each C2DM botnet traffic burst is also very small, in the range of 200-400 bytes. This shows the stealthiness property of a C2DM botnet in the data plane.

5.2 Efficiency in Resource Consumption

One major difference between a conventional PC-based botnet and a mobile botnet is that the latter needs to consider resource consumption, especially for bandwidth and power, because most mobile phones have limited network and battery capacities. If a mobile bot significantly consumes bandwidth or battery resources, then it may draw unnecessary attention of users and reveal the pres-

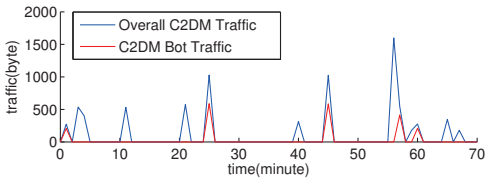


Figure 6: C2DM traffic generated by a C2DM bot and legitimate applications.

ence of a bot. We now evaluate and compare the bandwidth and power consumptions of a C2DM botnet with other mobile botnets that use traditional IRC and HTTP as C&C channels.

5.2.1 *Bandwidth Consumption*

We install a C2DM bot, an IRC bot and three HTTP bots (with time intervals for pulling commands from the C&C server as 5, 10, and 30 minutes) into five different Android phone emulators. We then measure the traffic consumption of each Android phone for one hour. To do a fair comparison, we compare the overhead in the control plane (in other words, we assume these bots have the same bandwidth consumption in disseminating commands). Thus, the traffic consumption of each bot depends on two factors: (a) the time interval of heartbeat connections, and (b) the volume of the traffic of each heartbeat connection. Figure 7 depicts the heartbeat traffic of these three types of bots.

• **C2DM bot:** The time interval of the heartbeat connections of the C2DM service is in the range of 15 to 30 minutes depending on the state of the mobile phone. In our experiment, the time interval is around 28 minutes. The traffic of each heartbeat message is between 250 and 300 bytes. Note that if the phone has are other legitimate applications which also use the C2DM service, then the bot will use the existing heartbeat connection and will not generate any extra heartbeat traffic.

• **IRC bot:** The IRC bot uses the *ping-pong mechanism* to keep alive, such that it sends a ping request to the C&C server and waits for the response from the C&C server. This can be treated as the heartbeat connection [28]. The time interval of the ping-pong conversations can be customized and is usually between 30 and 600 seconds as set in most of today's IRC server software. Note that the ping-pong interval cannot be too large, or it will impose heavy burden on the IRC server to maintain many persistent connections for a long time. In our experiment, the IRC bot is implemented based on PircBot [29], and the C&C server is built using the open-source IRC server software named beware-ircd [30] with default settings, where the ping-pong interval is 90 seconds and the traffic generated by each ping-pong conversation is around 200 bytes.

• **HTTP bot:** Unlike C2DM and IRC bots, a HTTP bot does not use any persistent TCP connection to keep alive with the C&C server, so it cannot receive commands instantly and has to connect to the C&C server from time to time to pull for commands. We regard this periodic pull-command connection as its heartbeat connection. The traffic of each pull-command connection is about 1000 bytes even when there is no command. This size is much larger than those of C2DM and IRC bots because the HTTP bot has to re-establish a TCP connection each time and transmit packets to complete the TCP 3-way handshake. HTTP packets are also larger than TCP packets in general due to the additional application-layer payload. In addition, the time interval of the pull-command connections affects the average delay of command dissemination. For example, if the time interval for pulling commands is set to X minutes, then the average time delay for each bot to receive a new command will be $X/2$ minutes. In our experiment, we set the bot to periodically retrieve commands from the C&C server using HTTP requests to imitate the pull-command behaviors of a HTTP bot.

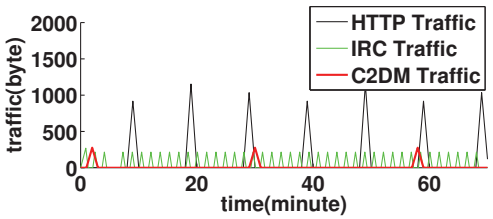


Figure 7: Heartbeat bandwidth consumption for C2DM, IRC and HTTP bots.

Figure 7 shows the result with the pull-command interval set to 10 minutes. We summarize the results here. The average traffic rate of the C2DM heartbeat connections is 900 bytes/hour, while those of the IRC and HTTP heartbeat connections are 5,760 bytes/hour (i.e., 6.4 times) and 7,200 bytes/hour (i.e., 8 times), respectively. This shows that a C2DM bot generates significantly less heartbeat traffic and hence bandwidth resources.

5.2.2 *Power Consumption*

We now evaluate and compare the power consumptions of C2DM, IRC, and HTTP bots. Our evaluation is built on [31], which propose power estimation models for different components on the basis of an Android HTC Dream phones. Here, we focus on the power consumption in the *communication* component, including WiFi and 3G, during the heartbeat period of each type of bots.

WiFi power consumption. To measure the WiFi power consumption, we install each bot in an Android phone emulator running on a PC, which is connected to a C&C server over a university WiFi network. We install Wireshark on the same PC, and capture all packets of the emulator in order to calculate the transmission time.

Figure 8 shows the power states of WiFi transmissions in an Android HTC phone. The WiFi power consumption depends on the number of packets sent/received per second, while the packet size has limited influence [31]. When a phone connects to a WiFi network and stays idle, it is in the "low" power state. When it sends/receives data, it switches to either the "ltransmit" or "htransmit" state, according to the transmission speed of the packets; when it finishes sending/receiving data, it will switch back to the previous state. In our experiment, the maximum packet speeds of all types of bots are below 15 packets per second, so the only involved power states during our experiments are "ltransmit" and "low" power states, as shown in Figure 8.

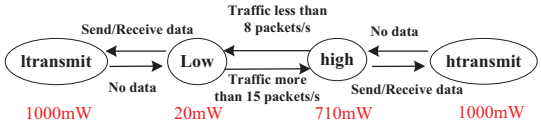


Figure 8: Wi-Fi power states of Android HTC Dream [31].

We calculate the power consumption (in Joules) during heartbeat transmission in one hour of each bot based on Equation (1), assuming that the total time span of transmitting packets[3] in one hour is T_l and the power of the "ltransmit" state is P_l:

$$W = P_l \cdot T_l. \quad (1)$$

[3]We do not consider the power consumption of the "low" power state since every phone which connects to the Internet with WiFi will stay in the "low" power state while idle.

Suppose that it takes time t (in seconds) for a bot to transmit all packets in each heartbeat connection. Let the heartbeat interval be λ (in seconds). We have:

$$T_l = t \cdot 3600 \cdot \lambda^{-1}. \quad (2)$$

Thus, we get:

$$W = P_l \cdot t \cdot 3600 \cdot \lambda^{-1}. \quad (3)$$

Table 1 shows the WiFi power consumption for heartbeat transmission of each bot in one hour, such that the inputs are based on our measurements (some results are obtained in §5.2.1). Note that for the HTTP bot, its packet transmission time t for each heartbeat connection is longer than those of C2DM and IRC bots, as it needs to set up a TCP connection with 3-way handshake. From Table 1, we observe that under the WiFi power model, the C2DM bot consumes the *least* amount of power resources compared to HTTP and IRC bots.

Type of Bot	t (in sec)	λ (in sec)	W (in mJ)
C2DM	0.1	1680	214.3
IRC	0.1	90	4000.0
HTTP (5min)	0.3	300	3600.0
HTTP (10min)	0.3	600	1800.0
HTTP (30min)	0.3	1800	600.0

Table 1: WiFi power consumption of each bot in one hour.

3G power consumption. To measure the 3G power consumption, we install each bot on a real Android phone that is connected to a C&C server over an operational 3G data network. We use tcpdump to capture all data packets within the Android phone.

Figure 9 shows the power states of 3G transmissions of an Android HTC phone. The 3G power consumption depends on the traffic volume. When the state is IDLE, the phone cannot send/receive any data through 3G, and it consumes almost no power. In the CELL_FACH state, it can send/receive a few hundred bytes of data per second, and waits for 6 seconds before switching back to the IDLE state. If the traffic rate is much larger than the transmission speed of the CELL_FACH state, then the power state will enter the CELL_DCH state, and the phone will wait for 4 seconds before switching back to the CELL_FACH state. The CELL_DCH has a higher transmission speed and higher power consumption than the CELL_DCH state.

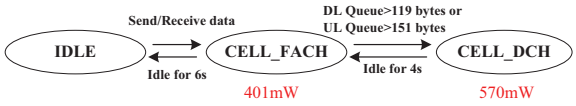


Figure 9: 3G power states of an Android HTC Dream [31].

We calculate the 3G power consumption during heartbeat transmission in one hour of each bot based on Equation (4), assuming that the power consumption in the IDLE state is zero, the data transmission time in CELL_FACH is T_f, the idle time in CELL_FACH is T_{fi}, the data transmission time in CELL_DCH is T_d, the idle time in CELL_DCH is T_{di}, the power in CELL_FACH is P_f, and the power in CELL_DCH is P_d:

$$W = P_f \cdot (T_f + T_{fi}) + P_d \cdot (T_d + T_{di}). \quad (4)$$

As in the WiFi analysis, we let t (in seconds) be the packet transmission time in each heartbeat connection, and λ (in seconds) be the interval of keep-alive connections. For C2DM and IRC bots, their heartbeat traffic volumes are generally small (see §5.2.1), so they remain in the CELL_FACH state (i.e., low power) during the heartbeat connections. Thus, the 3G power consumptions for C2DM and IRC bots are:

$$W = P_f \cdot (t + 6) \cdot 3600 \cdot \lambda^{-1}. \quad (5)$$

On the other hand, for the HTTP bot, its heartbeat traffic volume is larger (see §5.2.1), and it will stay in the CELL_FACH state (i.e., high power). Thus, the 3G power consumption of the HTTP bot is:

$$W = P_f \cdot 6 \cdot 3600 \cdot \lambda^{-1} + P_d \cdot (t + 4) \cdot 3600 \cdot \lambda^{-1}. \quad (6)$$

Table 2 shows the 3G power consumption for heartbeat transmission in one hour for each type of bots. Note that the packet transmission time t for the HTTP bot is about 1 second in 3G, which is larger than 0.3 seconds in WiFi, since the TCP round-trip delay is more significant in 3G than in WiFi. Similar to WiFi, the C2DM bot consumes the *least* amount of 3G power among all types of bot.

Type of Bot	t (in sec)	λ (in sec)	W (in mJ)
C2DM	0.1	1680	4892.2
IRC	0.1	90	97844.0
HTTP (5min)	1	300	63072.0
HTTP (10min)	1	600	31536.0
HTTP (30min)	1	1800	10512.0

Table 2: 3G power consumption of each bot in one hour.

5.3 Controllability

We now explore the following question: if the botmaster wants to send a command to N mobile bots, what is the minimum time T^* needed such that with a high probability p, all these N bots have received the command? We say that a botnet has good *controllability* if T^* is small for given N and p. Enabling good controllability is important for a botnet, for instance, when the botmaster wants to launch a synchronized jamming or DDoS attack. Here, we seek to show that a C2DM botnet can have good controllability.

Let $\hat{T}_k$ be the random variable of the time duration that a botmaster needs to send a notification to the k^{th} bot via the C2DM service. $\hat{T}_k$ can be expressed as:

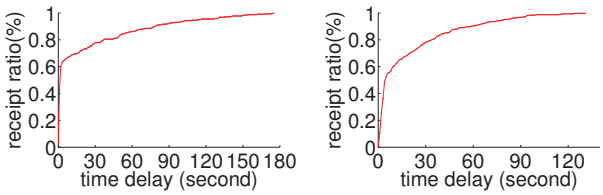
$$\hat{T}_k = (k-1)\delta + \hat{\tau}_k, \quad k = 1, \ldots, N, \quad (7)$$

where δ is the time between two consecutive C2DM requests generated by the botmaster, and $\hat{\tau}_k$ is the random variable of the time duration of sending a request from the botmaster via the C2DM service to the k^{th} bot. In our measurement, we set $\delta = 1/50$ seconds, implying that the botmaster generates 50 requests to the C2DM service per second. Note that this is significantly below our 100 QPS requirement to avoid alerting the C2DM service (see §4.3).

Note that $\hat{\tau}_k$ is related to the network conditions, phone status, C2DM server load, etc. We perform two experiments to measure the probability distribution of $\hat{\tau}_i$ in both WiFi and 3G networks, using the similar setups in §5.2.1. Figure 10(a) shows the measurement results for the WiFi experiment. We find that in 60% of time, a command is delivered within 2 seconds; and in 95% of time, a command is delivered within 2 minutes. Figure 10(b) shows the measurement results for the 3G experiment. In 53% of time, a command is delivered within 5 seconds; in 99% of time, it can be delivered in 2 minutes.

Let $\hat{T}$ be the random variable for *all* bots to receive the same command from the botmaster. We can express $\hat{T}$ as:

$$\hat{T} = \max_{i \in \{1,..,N\}} \{\hat{T}_k\} = (N-1)\delta + \max_{i \in \{1,..,N\}} \{\hat{\tau}_k\}. \quad (8)$$



(a) t in WiFi (b) t in 3G

Figure 10: Probability distribution function of time delay $\hat{\tau}$ for bots to receive a C2DM message: (a) WiFi (b) 3G.

To ease our analysis, we approximate $\hat{\tau}_k$ as an independent and identically distributed exponential random variable with parameter μ, which denotes the average command arrival rate to a bot. Let f_{τ^*} be the probability density function of $\max_{i\in\{1,..,N\}}\{\hat{\tau}_k\}$. Using order statistics [32], we have:

$$f_{\tau^*}(t) = N\left(1-e^{-\mu t}\right)^{N-1}\mu e^{-\mu t} \quad \text{for } t \geq 0. \tag{9}$$

To find T^* such that with a high probability p all N bots have received the command, we can numerically evaluate the following expression to obtain T_d:

$$\int_{t=0}^{T_d} f_{\tau^*}(t) = p. \tag{10}$$

Finally, $T^* = (N-1)\delta + T_d$. To illustrate, we consider a C2DM botnet with $N = 10000$ bots, $\delta = 1/50$ seconds, $1/\mu$ is 21.7 seconds for WiFi, and 18.9 seconds for 3G (the latter two are based on our measurements in Figures 10(a) and 10(b), respectively). Table 3 shows different values of T^* under different probability requirements p. In summary, it takes less than 8 minutes to reach all bots in a large-scale C2DM botnet. This shows the good controllability of a C2DM botnet.

p	T^* (WiFi)	T^* (3G)
0.80	432.4 sec	402.4 sec
0.90	448.7 sec	416.6 sec
0.95	464.3 sec	430.2 sec

Table 3: Controllability: value of T^* under different probability guarantees.

6. BOTNET DEPLOYMENT AND ATTACK

In this section, we present the details of deploying a C2DM botnet from an attacker's perspective. As a proof of concept, we also implement a small-scale C2DM botnet. We demonstrate how it can be injected into legitimate mobile applications and used to launch a real-life spamming attack.

6.1 Deployment

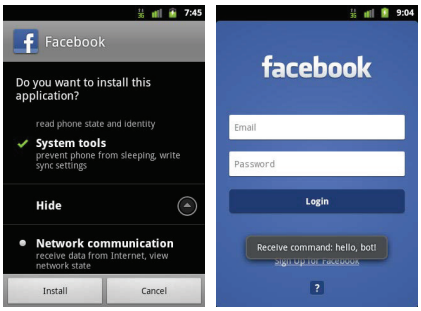
Infection. We first explain how we infect a target Android application with a C2DM bot. Note that the infection approach we describe here is also applicable for adding any types of malicious code.

A typical Android application is composed of multiple object files and packaged into a single file with .apk extension. Each object file contains executable bytecode designed for the runtime environment called the Dalvik Virtual Machine. To inject malicious code into a target Android application, one cannot directly operate on the application's source code that is generally unavailable. Instead, one can *disassemble* the bytecode of the target application into assembly-like code called *Smali* code, using the official Android Apktool software [33].

After disassembling an Android application, one can access a manifest file called AndroidManifest.xml, which describes the metadata of an Android application, including the name, permissions, activities, and services. It also specifies the *main activity*, which is the first activity that will start when the application is launched, with a tag called "android.intent.action.MAIN". An attacker can modify the tag to change the main activity of the target application to start the malicious activity (i.e., the C2DM bot program). The malicious activity should run in the background so that the infected application appears to behave normally. In addition, the attacker needs to modify AndroidManifest.xml to add extra permissions for the malicious activity. For the C2DM bot, we add the admission "com.google.android.c2dm.permission.RECEIVE".

Propagation. One way to propagate the infected target application is to upload it to the official Android Market, but the infected application may be blocked due to the checking procedure. An attacker can upload the infected application to some third-party markets such as HiAPK and AppChina. In China and some Middle East countries, the Android Market may have low download bandwidth or may be blocked [34], so users may have to rely on third-party markets to download applications. This makes the propagation of an infected application feasible.

Implementation. We implement a proof-of-concept C2DM bot prototype and inject it into a popular application called Facebook for Android, which uses many permissions including the C2DM access, full Internet access, read contact data, and GPS. This feature enables us to inject the C2DM bot into the application without requiring extra permissions. First, we note that the permission "com.google.android.c2dm.permission.RECEIVE" for the C2DM service is not as sensitive as other permissions, such as sending SMS and receiving SMS, in drawing attention of users. During the installation, this permission is represented as "receive data from Internet" (see Figure 11(a)), which is hidden in a collapsed list. Users may not notice this permission is included. Figure 11(b) shows that once the application is launched, the C2DM bot service will start in background and it can receive commands from the botmaster via the C2DM service.



(a) Installation (b) Startup

Figure 11: Injecting a C2DM bot into the Facebook app.

6.2 Attack Demonstration

We conduct a trial study on deploying a C2DM botnet in reality and launching a spamming attack known as the *SMS-spam-and-click* attack. Our goal is to demonstrate the threat of a mobile botnet in practice, using our C2DM botnet as a motivating example.

Specifically, we recruited 10 volunteers and installed C2DM bots into their Android phones. Thus, we construct a real-life, small-

scale C2DM botnet with 10 bots. These bots will generate an SMS message containing a URL that refers to a website under our control for data collection. We then experiment three different forms of attacks:

• **Random spamming:** Each of the 10 bots sends 10 SMS messages to 10 different registered phone numbers selected at random. We are interested in finding how many people will click our URL link. Note that this attack is similar to the conventional spamming attack. In our experiment, none of the 100 people clicked our link, which means that the conventional spamming attack may not receive a high number of clicks.

• **Contacts spamming:** Each of the 10 bots now sends an SMS message to 10 known people who are in the phone's contact list. The result is that 23 out of 100 people clicked our URL, and this shows that exploiting friendship in an attack can enhance the click success rate.

• **Man-in-the-middle spamming:** Each of the 10 bots monitors the SMS conversation of the compromised mobile phone, and the bot sends an SMS message to the contacted person on the other end. We instruct each bot to send a maximum of 10 SMS messages. Our experiments show that 45% of contacted persons clicked the URL. This shows that the man-in-the-middle spamming can also have a high success rate.

7. EXTENSION TO OTHER PLATFORMS

We mention in §1 that many platforms other than Android provide push notification services and can be exploited as well. We now elaborate how to extend a C2DM botnet to other platforms.

The push notification services of other platforms are similar to C2DM in the architectural design, as shown in Figure 12. When an application launches in a mobile device, it needs to register to the push service to get a unique ID (it may have different names in different platforms, e.g., *device token* in iOS and *push URI* in Windows), and then sends it to the application server. When the application server wants to send a push notification to an application, it sends the ID together with the payload to a push server, which then forwards the payload to the application.

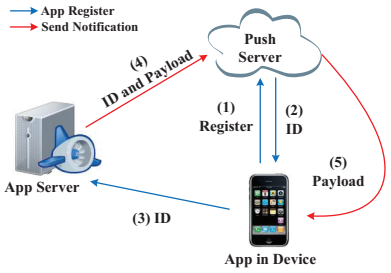


Figure 12: Architecture of push notification service.

Table 4 compares different push notification services, including Google's C2DM, Apple's Push Notification Service (APNS), Microsoft's Push Notification Service (MPNS), Blackberry's Push Service (BPS), and Nokia's Notification API (NNA). The maximum payload size of APNS is 256 bytes, which is the smallest among all, but it still suffices for a typical botnet command. In terms of request quota, for APNS and NNA, they do not set any limit on the number of push notifications that the application server can send per day. For MPNS, the quota is unlimited for an authenticated web service, and 500 per day per device for unauthenticated ones. To authenticate a web service, a developer needs to apply for a certificate from a certificate authority and upload it to Windows Phone Marketplace. For BPS, it has two versions: Essential and Plus. Their difference is that the Plus version supports delivery status report. The quotas of the Essential and Plus versions are unlimited and 100,000 for free per day. Thus, compared to the C2DM platform, botnets on other platforms in general have better scalability due to less strict quota limits.

Service	Max. Payload	Quota (per Day)
C2DM	1 KB	200,000
APNS	256 Bytes	Unlimited
MPNS	3 KB (+ 1KB Header)	Unlimited (authenticated) 500 (unauthenticated)
BPS	8 KB	Unlimited (Essential version) 100,000 free (Plus version)
NNA	1.5 KB	Unlimited

Table 4: Comparison of push notification services.

Similar to C2DM, each push request of other push notification services can be sent to one device only (see §4.1). Thus, the controllability of a botnet in other platforms is also determined by the QPS limitation, as discussed in §5.3.

Note that the C2DM service is deprecated on June 26, 2012, and is replaced by the *Google Cloud Messaging for Android (GCM)* [35]. GCM is a push notification service that is based on C2DM, with several enhancements: supporting multicast messages, no quotas, supporting JSON-based notification messages, better battery efficiency, etc. Nevertheless, the fundamental design of the C2DM botnet is to exploit the push notification service as a C&C channel, and we expect that our proposed mobile botnet design remains applicable in GCM. In fact, the GCM botnet might potentially be more effective in command dissemination as a command can be sent to multiple bots via multicast, and no quota is imposed. We plan to further investigate it in future work.

In summary, we believe that it is feasible to extend the C2DM botnet design to other push notification platforms. Furthermore, the botnets on such platforms share the same properties as the C2DM botnet, such as stealthiness, resource efficiency, and controllability.

8. RECOMMENDATIONS ON DEFENSE

In this section, we discuss possible defense strategies against a C2DM botnet, or more generally, push-styled mobile botnets deployable in today's push notification platforms. While we have yet identified this kind of botnets in the wild, our work suggests that their eventual existence is anticipated.

We point out that most existing botnet detection methods cannot effectively detect push-styled mobile botnets. Since both push-styled bots and legitimate applications connect to official push notification servers to receive messages, it is non-trivial for anomaly-based detection methods (e.g., BotSniffer [22]) and mining-based detection methods (e.g., BotMiner [23]) to separate botnet traffic from other legitimate application traffic. In the following, we suggest possible defense strategies against push-styled mobile botnets.

Based on our C2DM botnet design, a bot will send its unique ID to the botmaster directly, or via push notification servers, during its registration. Thus, we may deploy a sandbox system in a mobile device to monitor the network behavior of an application. The sandbox system can check whether an application sends data to suspicious addresses or sends suspicious requests to push notification servers. Since many legitimate applications also send data to legitimate addresses or push notification servers, this defense mechanism requires further traffic analysis to confirm the existence of a push-styled bot.

Our experience is that not every mobile device platform has strict

authority management over push notifications. Users may unnecessarily enable certain applications to support push notification services. Specifically, in Android, the permission of receiving C2DM messages "com.google.android.c2dm.permission.RECEIVE" is not a *native permission*, which starts with "android.permission.". Thus, it has not been considered as sensitive as other native permissions which may incur monetary costs to users or require access to private information, such as "android.permission.SEND_SMS" and "android.permission.RECEIVE_SMS". We suggest that the permission of receiving C2DM messages should also be treated as one of the sensitive permissions. If an application requires this permission, then it should be examined to ensure that the permission is actually required for its functions. In addition, the manifest file AndroidManifest.xml of the application should be checked to see whether there is more than one C2DM receiver. In this case, the application may have been injected with some unauthorized C2DM receivers.

9. CONCLUSION

Push notification services have provided great convenience and flexibility for applications to receive light messages from application servers. In this paper, we propose the design of a novel cloud-based push-styled mobile botnet, which exploits the push notification service for lightweight command dissemination. We take Google's C2DM service for the Android platform as a motivating example to construct this type of mobile botnets. We demonstrate how a C2DM botnet can feasibly utilize Google's C2DM service as a C&C channel, and justify how its design can be made scalable and resilient against service disruption. Through in-depth evaluation, we show that the C2DM botnet is stealthy, resource-efficient, and controllable. We prototype the C2DM botnet and demonstrate how it can be deployed in reality. Finally, we provide recommendations on potential defense strategies against such push-styled mobile botnets in general. Our C2DM botnet prototype is available for download at **http://ansrlab.cse.cuhk.edu.hk/software/c2dmbotnet**.

10. ACKNOWLEDGMENTS

The work is supported in part by the National Natural Science Foundation of China under grants 60921003 and 61103241.

11. REFERENCES

[1] A. Apvrille. Symbian worm yxes: Towards mobile botnets? In *19th Annual EICAR Conference, France*, 2010.

[2] P. Porras, H. Saïdi, and V. Yegneswaran. An analysis of the ikee. b iphone botnet. *Security and Privacy in Mobile Information and Communication Systems*, pages 141–152, 2010.

[3] Lookout Inc. Security alert: Geinimi, sophisticated new android trojan found in wild, 2010. http://blog.mylookout.com/blog/2010/12/29/geinimi_trojan.

[4] Trend Micro Inc. Zeus targets mobile users. http://blog.trendmicro.com/zeus-targets-mobile-users, 2011.

[5] X. Jiang. Security Alert: AnserverBot, New Sophisticated Android Bot Found in Alternative Android Markets. http://www.csc.ncsu.edu/faculty/jiang/AnserverBot/, Sep 2011.

[6] Kaspersky Inc. Irc bot for android. http://www.securelist.com/en/blog/208193332/IRC_bot_for_Android, 2012.

[7] Google Inc. Android Cloud to Device Messaging Framework. http://code.google.com/android/c2dm.

[8] Apple Inc. Local and Push Notification Programming Guide. http://developer.apple.com/library/mac/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/RemoteNotificationsPG.pdf, 2011.

[9] Microsoft Inc. Push Notifications Overview for Windows Phone. http://msdn.microsoft.com/en-us/library/ff402558(v=vs.92).aspx.

[10] Reserach In Motion Inc. Blackberry push service. http://http://us.blackberry.com/developers/platform/pushapi.jsp.

[11] Nokia Inc. Notifications api. https://projects.developer.nokia.com/notificationsapi/wiki.

[12] P. Traynor, M. Lin, M. Ongtang, V. Rao, T. Jaeger, P. McDaniel, and T. La Porta. On Cellular Botnets: Measuring the Impact of Malicious Devices on a Cellular Network Core. In *Proc. of ACM CCS*, 2009.

[13] K. Singh, S. Sangal, N. Jain, P. Traynor, and W. Lee. Evaluating Bluetooth as a Medium for Botnet Command and Control. In *Proc. of DIMVA*, 2010.

[14] C. Xiang, F. Binxing, Y. Lihua, L. Xiaoyi, and Z. Tianning. Andbot: Towards Advanced Mobile Botnets. In *Proc. of USENIX LEET*, pages 11–11. USENIX Association, 2011.

[15] K.G. Zeng, Y.and Shin and X. Hu. Design of SMS Commanded-and-Controlled and P2P-structured Mobile Botnet. In *Proc. of ACM WiSec*, 2012.

[16] G. Geng, G. Xu, M. Zhang, Y. Yang, and G. Yang. An improved sms based heterogeneous mobile botnet model. In *Information and Automation (ICIA), 2011 IEEE International Conference on*, pages 198–202. IEEE, 2011.

[17] J. Hua and K. Sakurai. A SMS-based Mobile Botnet Using Flooding Algorithm. In *Proc. of IFIP WISTP*, 2011.

[18] G. Weidman. Transparent Botnet Command and Control for Smartphones over SMS. In *Shmoocon*, 2011.

[19] C. Mulliner and J.P. Seifert. Rise of the iBots: 0wning a Telco Network. In *Proc. of IEEE MALWARE*, pages 19–20, 2010.

[20] M. Akiyama, T. Kawamoto, M. Shimamura, T. Yokoyama, Y. Kadobayashi, and S. Yamaguchi. A Proposal of Metrics for Botnet Detection Based on its Cooperative Behavior. In *Proc. of SAINT Workshops*, pages 82–82. Ieee, 2007.

[21] H. Choi, H. Lee, H. Lee, and H. Kim. Botnet Detection by Monitoring Group Activities in DNS Traffic. In *Proc. of IEEE CIT*, 2007.

[22] G. Gu, J. Zhang, and W. Lee. BotSniffer: Detecting Botnet Command and Control Channels in Network Traffic. In *Proc. of NDSS*, 2008.

[23] G. Gu, R. Perdisci, J. Zhang, and W. Lee. BotMiner: Clustering Analysis of Network Traffic for Protocol-and Structure-Independent Botnet Detection. In *Proc. of USENIX Security*, 2008.

[24] I. Vural and H. Venter. Mobile Botnet Detection Using Network Forensics. In *Future Internet Symposium*, 2010.

[25] E. Kartaltepe, J. Morales, S. Xu, and R. Sandhu. Social Network-Based Botnet Command-and-Control: Emerging Threats and Countermeasures. In *Proc. of ACNS*, 2010.

[26] Android Developers. http://developer.android.com.

[27] Tim Hopper. My email analytics. http://www.stiglerdiet.com/2012/04/05/my-email-analytics/.

[28] J. Oikarinen and D. Reed. Internet relay chat protocol. *RFC 1459*, 1993.

[29] P. Mutton. Pircbot 1.2. 5 java irc api: Have fun with java. *Java Developer's Journal*, 8(12):26–32, 2003.

[30] Beware ircd. http://ircd.bircd.org.

[31] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R.P. Dick, Z.M. Mao, and L. Yang. Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones. In *Proc. of ACM CODES+ISSS*, pages 105–114. ACM, 2010.

[32] Herbert A. David. *Order Statistics, 2nd Ed.* Wiley-Interscience, 1981.

[33] Apktool. http://code.google.com/p/android-apktool/.

[34] S. Ye. Android Market is Currently Blocked in China. Here are your Alternatives, Sep 2011. http://techrice.com/2011/10/09/android-market-is-currently-blocked-in-china-here-are-your-alternatives/.

[35] Google Inc. Migration. http://developer.android.com/guide/google/gcm/c2dm.html, 2012.

DISCLOSURE: Detecting Botnet Command and Control Servers Through Large-Scale NetFlow Analysis

Leyla Bilge
Symantec Research Labs
leylya_yumer@symantec.com

Davide Balzarotti
Eurecom
balzarotti@eurecom.fr

William Robertson
Northeastern University
wkr@ccs.neu.edu

Engin Kirda
Northeastern University
ek@ccs.neu.edu

Christopher Kruegel
UC Santa Barbara
chris@cs.ucsb.edu

ABSTRACT

Botnets continue to be a significant problem on the Internet. Accordingly, a great deal of research has focused on methods for detecting and mitigating the effects of botnets. Two of the primary factors preventing the development of effective large-scale, wide-area botnet detection systems are seemingly contradictory. On the one hand, technical and administrative restrictions result in a general unavailability of raw network data that would facilitate botnet detection on a large scale. On the other hand, were this data available, real-time processing at that scale would be a formidable challenge. In contrast to raw network data, NetFlow data is widely available. However, NetFlow data imposes several challenges for performing accurate botnet detection.

In this paper, we present DISCLOSURE, a large-scale, wide-area botnet detection system that incorporates a combination of novel techniques to overcome the challenges imposed by the use of NetFlow data. In particular, we identify several groups of features that allow DISCLOSURE to reliably distinguish C&C channels from benign traffic using NetFlow records (i.e., flow sizes, client access patterns, and temporal behavior). To reduce DISCLOSURE's false positive rate, we incorporate a number of external reputation scores into our system's detection procedure. Finally, we provide an extensive evaluation of DISCLOSURE over two large, real-world networks. Our evaluation demonstrates that DISCLOSURE is able to perform real-time detection of botnet C&C channels over datasets on the order of billions of flows per day.

1. INTRODUCTION

Malware continues to run rampant across the Internet, and among the myriad forms that modern malware can assume, botnets represent one of the gravest threats to Internet security. Through the large-scale compromise of vulnerable end hosts, botmasters can both violate the confidentiality of sensitive user information—for instance, banking or social network authentication credentials—as well as leverage groups of bots as an underground computational platform for performing other illicit activities.

Accordingly, a great deal of research has focused on methods for detecting and mitigating the deleterious effects of botnets. Research to date has largely followed one of two major approaches: (i) vertical correlation, or detecting command and control (C&C) channels used by botmasters to communicate with each infected machine [17,19,21,29]; and (ii) horizontal correlation, where botnet detection is based upon patterns of crowd behavior exhibited by collections of bots in response to botmaster commands [6,16,18,32, 35]. Once bots or, ideally, C&C servers have been identified, a number of actions can be performed, ranging from removal of infected endpoints from the network, to filtering C&C channels at edge routers, to orchestrated take-downs of the C&C servers themselves.

Unfortunately, while previous botnet detection approaches are effective under certain circumstances, none of these approaches scales beyond a single administrative domain while retaining useful detection accuracy. This limitation restricts the application of automated botnet detection systems to those entities that are informed or motivated enough to deploy them. Thus, we have the current state of botnet mitigation, where small pockets of the Internet are fairly well protected against infection while the majority of endpoints remain vulnerable.

This situation is not ideal. Botnets are an Internet-wide problem that spans individual administrative domains and, therefore, a problem that requires an Internet-scale solution. In particular, botnets can continue to wreak havoc upon the Internet despite the deployment of localized detection systems by focusing on propagation through less well-protected populations.

Two of the primary factors preventing the development of effective large-scale, wide-area botnet detection systems are seemingly contradictory. On the one hand, technical and administrative restrictions result in a general unavailability of raw network data that would facilitate botnet detection on a large scale. On the other hand, were this data available, real-time processing at that scale would be a formidable challenge. While the ideal data source for large-scale botnet detection does not currently exist, there is, however, an alternative data source that is widely available today: NetFlow data [10].

NetFlow data is often captured by large ISPs using a distributed set of collectors for auditing and performance monitoring across backbone networks. While it is otherwise extremely attractive, NetFlow data imposes several challenges for performing accurate botnet detection. First, and perhaps most critically, NetFlow records do not include packet payloads; rather, flow records are limited to aggregate metadata concerning a network flow such as the flow duration and number of bytes transferred. Second, NetFlow records are half-duplex; that is, they only record one direction of a network connection. Third, NetFlow data is often collected by sampling the monitored network, often at rates several orders of magnitude or more removed from real traffic.

Each of these characteristics of NetFlow data complicates the development of an effective botnet detector over this domain. The detector must be able to distinguish between benign and malicious network traffic *without* access to network payloads, which is the component of network data that carries direct evidence of malicious behavior. The detector must also be able to recognize weak

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

signals indicating the presence of a botnet due to the combined effects of half-duplex capture and aggressive sampling.

In this paper, we present DISCLOSURE, a large-scale, wide-area botnet detection system that incorporates a combination of novel techniques to overcome the challenges imposed by the use of NetFlow data. In particular, we identify several groups of features that allow DISCLOSURE to reliably distinguish C&C channels from benign traffic using NetFlow records: (i) flow sizes, (ii) client access patterns, and (iii) temporal behavior. We demonstrate that these features are not only effective in detecting current C&C channels, but that these features are relatively robust against expected countermeasures future botnets might deploy against our system. Furthermore, these features are oblivious to the specific structure of known botnet C&C protocols.

While the aforementioned features are sufficient to capture core characteristics of generic C&C traffic, they also generate false positives in isolation. To reduce DISCLOSURE's false positive rate, we incorporate a number of external reputation scores into our system's detection procedure. These additional signals function as a filter that reduces DISCLOSURE's false positive rate to a level where the system can feasibly be deployed on large-scale networks.

We provide an extensive evaluation of DISCLOSURE over two real-world networks: a university network spanning a small country where no NetFlow sampling occurred, and a Tier 1 ISP where NetFlow data was sampled at a rate of one out of every ten thousand flows. Our evaluation demonstrates that DISCLOSURE is able to perform real-time detection of botnet C&C channels over data sets on the order of billions of flows per day. In particular, we show that DISCLOSURE is able to recognize approximately 65% of known botnet C&C servers in both settings while producing 1% false positives. Furthermore, we demonstrate DISCLOSURE's ability to detect previously unknown C&C servers by manually verifying 20 and 91 true positive alerts from the university and ISP networks, respectively.

Finally, we report on our operational experience in deploying and testing DISCLOSURE on real large-scale networks, highlighting the most critical areas for tuning the performance of the detection system. The contributions of the paper is as follows:

- We present DISCLOSURE, a large-scale, wide-area botnet detection system that reliably detects botnet C&C channels in readily-available NetFlow data using a set of robust statistical features. To our knowledge, DISCLOSURE is the only NetFlow-based system that does not assume *a priori* knowledge of particular C&C protocols.
- We incorporate several external reputation systems into DISCLOSURE's detection procedure to further refine the accuracy of the system.
- We evaluate DISCLOSURE over two real-world networks, and demonstrate its ability to detect both known and unknown botnet C&C servers at scales not previously achieved.
- We report on our operational experience with DISCLOSURE, and highlight important tuning considerations for deployment and reproducibility.

2. SYSTEM OVERVIEW

DISCLOSURE is a botnet detection system designed to identify C&C servers by employing NetFlow analysis. Figure 1 shows an overview of the system architecture. The upper half of the figure describes the detection model generation process, where a supervised machine learning algorithm is used to train models on a subset of NetFlows targeting known (i.e., labeled) benign and C&C servers.

The flows in this labeled data set are first processed by the *feature extraction module*. This module reduces the flows to a number of distinct features: flow size-based features, client access pattern-based features, and temporal features, which are described in detail in Section 3. The features extracted from the training set are then forwarded to DISCLOSURE's *learning module*, which is responsible for building detection models. The learning module can be tuned with several thresholds to obtain an optimal balance between detection and false positive rates.

The bottom half of the graph represents the detection phase, where the models that have been previously generated are applied to unlabeled NetFlows in order to distinguish benign traffic from C&C communication. Since the aim of DISCLOSURE is not to identify bot-infected machines but to detect C&C servers, the first task of the detection phase is to filter those NetFlows that cannot be attributed to a server; this process is explained in Section 5.4. Then, the flows are forwarded to the *feature extraction module*. Finally, the resulting feature vectors are processed by the *detection module* to produce the final list of suspected C&C servers.

Note that the results of DISCLOSURE can be further processed by a *false positive reduction* filter. The goal of this additional module is to correlate the results of DISCLOSURE with the information obtained from other security feeds in order to reduce the probability of misclassification. For example, in Section 4, we present a novel technique that associates a reputation score to the autonomous systems to which the C&C servers belong.

3. FEATURE SELECTION AND CLASSIFICATION

In this section, we present the features extracted by DISCLOSURE from NetFlow data in order to detect botnet C&C channels at scale, and discuss why these features are suitable for discriminating between C&C channels and benign traffic. We then describe the particular machine learning techniques we use to build detection models over these features.

3.1 NetFlow Attributes

NetFlow is a network protocol proposed and implemented by Cisco Systems [10] for summarizing network traffic as a collection of network flows. Network elements such as routers and switches capture these NetFlows and forward them to NetFlow collectors. A network flow is defined to be a unidirectional sequence of packets that share specific network properties (e.g., IP source/destination addresses, and TCP or UDP source/destination ports). Each flow has a number of associated attributes, or summary statistics that characterize various general aspects of its behavior. In this paper, the NetFlow attributes we analyzed for extracting features to identify C&C servers are: the source IP address, the destination IP address, the TCP source port, the TCP destination port, the start and finish timestamps, and the number of bytes and packets transferred.

Since DISCLOSURE is primarily focused on identifying botnet C&C channels, it is imperative that the system can reliably distinguish servers from clients. Therefore, as an intermediate preprocessing stage, NetFlow data is analyzed by the server classifier that labels each observed IP address according to whether it provides one or more network services. In particular, since multiple services can be made available for each IP address, we represent each server as a 2-tuple of IP address and port, $s_i = \langle a, p \rangle$, where $a \in A$ is an IP address, A is the set of all IP addresses, $p \in P$ is a TCP port, and P is the set of all ports.

A common, and legitimate, criticism of early attempts to perform machine learning-based detection over NetFlow data is that the features that were selected were often not *robust*. Hence, the resulting detection systems would often overfit models to the specific behavior of malware represented in the training set—for instance, the particular server port used by a given malware sample. Such features, however, do not generalize to classes of malware such as botnets. For example, using our example of learning a model on server ports, it is clear that the use of a particular server port is not an intrinsic property of botnet behavior. Therefore, the design of DISCLOSURE's feature extractor module emphasizes the selection of those NetFlow attributes that best capture invariants in botnet behavior without resorting to specialization to a particular C&C protocol.

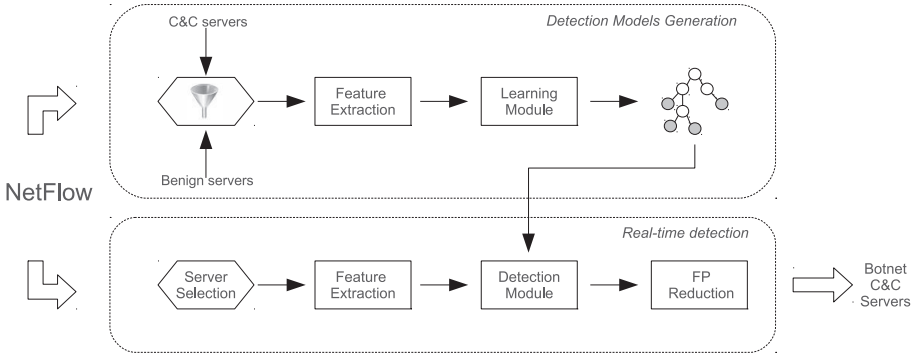


Figure 1: The system architecture of Disclosure. In the training phase (upper half), labeled training samples are used to build detection models. In the detection phase (lower half), the detection models are used to classify IP addresses as benign or associated with C&C communications.

3.2 Disclosure Feature Extraction

3.2.1 Flow Size-Based Features

The first class of features extracted from NetFlow data are based on *flow sizes*, which simply indicate the total number of bytes transferred in one direction between two endpoints for a particular flow. Our premise for analyzing flow sizes in NetFlow data is that the flow size distributions for C&C servers are significantly and *necessarily* different from flow size distributions for benign servers.

We attribute this difference to several factors. First, the main role of the botnet C&C channel is to establish a connection between the bots and the C&C server. This channel should be both reliable as well as relatively innocuous in appearance. Thus, flows carrying botnet commands or information harvested from infected clients are preferred to be as short as possible in order to minimize their observable impact on the network. Considering that network monitoring tools are widely used and that a botnet's local network impact usually scales linearly with the number of bot infections, tuning for stealth is an important goal. Moreover, due to the limited number of commands in typical C&C protocols, flow sizes tend not to fluctuate significantly. On the other hand, flow sizes generated during accesses to a benign server usually assume a wide range of values.

The preliminary analysis we performed on known sets of benign servers and C&C servers supports our premise. Hence, we designed a set of methods to extract features to detect the behavioral difference between C&C servers and benign servers with respect to flow size.

DISCLOSURE extracts flow size-based features by first grouping all flows according to the server s_i that they originate from or are destined to. Let $s_i \in S$ be a server, and $c_j \in C$ be a client. Then, flow sizes are grouped by time intervals $j = 0, 1, 2, \ldots$, where $F_{i,j}$ denotes a series of flow sizes for flows from endpoint i to j, where endpoints can be drawn from C or S. Once this set has been derived, the following feature sets are extracted.

Statistical features. This group of features characterizes the regularity of flow size behavior over time for both benign and C&C servers. In particular, we extract the mean $\mu^{F_{i,j}}$ and standard deviation $\sigma^{F_{i,j}}$ separately for both incoming and outgoing flows of each server.

Autocorrelation features. Autocorrelation is widely used for cross-correlating a signal with itself in the signal processing domain [7], and is useful for identifying repeating patterns in time series data. A series of flow sizes $F_{i,j}$ can be converted to a time series by ordering sizes by time. Since the autocorrelation function also requires a time series that is sampled periodically as input, we segment the time series by fixed intervals and take the mean over each interval; empirical testing suggested that a period of 300 seconds is appropriate. Once a periodically sampled time series $\hat{F}_{i,j}$ has been derived from $F_{i,j}$, the series is processed by the autocorrelation function, and features are extracted from the output. Here, we use a discrete autocorrelation coefficient $R_{\hat{F}_{i,j}\hat{F}_{i,j}}$ with lag j normalized by the variance σ^2, where

$$R_{\hat{F}_{i,j}\hat{F}_{i,j}} = \frac{\sum_{i=j}^{n} x_i \overline{x}_{i-j}}{\sigma^2}.$$

The autocorrelation function outputs the correlation results for each period in the time series. This output is further processed by taking the mean and standard deviation over these values to derive the final autocorrelation features.

Unique flow sizes. In addition to the statistical features described above, DISCLOSURE also includes features that count the number of unique flow sizes observed, and performs statistical measurements of occurrence density for each of them during the analysis time. Specifically, an array is constructed in which the elements are the number of occurrences of a specific flow size. Afterward, statistical features are computed over this flow size incidence array to measure its regularity.

3.2.2 Client Access Patterns-Based Features

One typical property of botnets is that the bots frequently establish a connection with the C&C server. These connections tend to be ephemeral, as longer-lived connections might draw undue attention to a bot's presence.

Our basis for selecting features to extract in order to distinguish malicious client access patterns from benign ones is that all of the clients of a C&C server (i.e., bots) should exhibit similar access patterns, whereas the clients of a benign server should not. Since all bots share the same, or nearly identical, malicious program, they tend to access C&C servers similarly unless specifically programmed otherwise. On the other hand, clients of benign services tend to exhibit much more varied patterns due to the vagaries of human action. DISCLOSURE extracts two sets of features to characterize client access patterns typical of C&C servers and those typical of benign servers.

Regular access patterns. For each server s_i and client c_j, DISCLOSURE prepares a time series $T_{i,j}$ of flows observed during the analysis period. Then, a sequence of flow inter-arrival times $I_{i,j}$ is derived from the time series by taking the difference between consecutive connections; that is,

$$I_{i,j} = \bigcup_{k=1}^{n} t_{i,j,k} - t_{i,j,k-1},$$

where $t_{i,j,k}$ is the k^{th} element of $T_{i,j}$. Then, statistical features are computed over each inter-arrival sequence, including the minimum, maximum, median, and standard deviation. Finally, we derive the

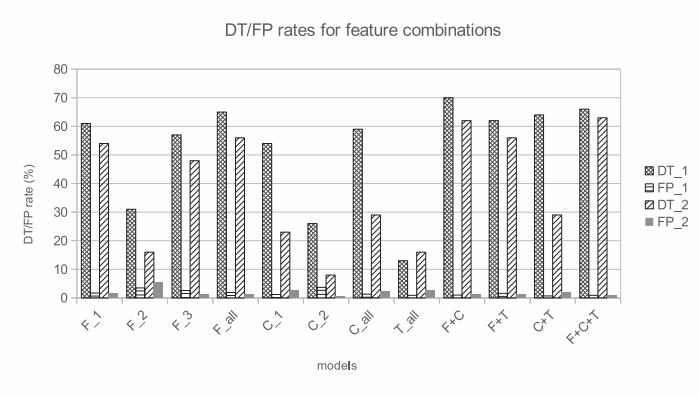


Figure 2: Detection rates (DT) and false positive (FP) rates for different feature combinations. We note that the DT:FP ratio is most favorable when all features are used in the detection procedure.

final features for each server s_i by generating statistical features across the set of clients that accessed s_i. This allows DISCLOSURE to not only find regular patterns in clients, but to determine whether the set of clients accessing a server behave similarly.

Unmatched flow density. When a bot is unable to communicate with a legitimate C&C server, it detaches from the rest of the botnet and becomes a zombie. This might happen because the C&C server was shutdown, or its IP address has been blacklisted. Since the zombie cannot distinguish between these situations and transient network errors, it continues querying the server. This can result in a significant number of flows to a server that do not have a matching flow in the opposite direction. It is also possible that a benign server is unreachable for a period of time. However, the behavior of a benign server's clients is significantly different than the behavior of bots that lose access to their C&C servers. This is because when a benign user is aware that a server is offline, it typically does not insist on continuing to query the server indefinitely. Therefore, DISCLOSURE extracts statistics regarding the number of unmatched incoming and outgoing flows to detect this behavior. Specifically, let $U_{i,j}$ be the number of unmatched flows for server s_i in time interval t_j, where

$$U_{i,j} = \sum_{j \in C} \text{abs}\left(|F_{i,j}| - |F_{j,i}|\right).$$

Then, DISCLOSURE derives the mean and standard deviation over a time series of $U_{i,j}$ as a statistical feature.

3.2.3 Temporal Features

Connections to a benign server are subject to diurnal fluctuations representative of the server's user population. On the other hand, connections to C&C servers are dictated by the botmaster, and require no user intervention. As previously mentioned, the majority of botnets configure their bots to contact the C&C server periodically and with relatively short intervals. Therefore, bot-infected machines connect to C&C servers during periods of the day that benign clients do not. For example, many benign servers receive a high volume of traffic during the day, and very little—or nothing—during the night.

To capitalize on this observation, DISCLOSURE extracts a set of temporal features that characterize the variability of client flow volume as a function of time, such that the system can discriminate between uniform client flow distributions indicative of C&C servers and benign traffic that follows well-known diurnal patterns. Specifically, DISCLOSURE segments a time series of client and flow volume by hour-long intervals per server s_i, and calculates statistical features over these.

3.3 Building the Detection Models

To build detection models for identifying C&C servers, we experimented with a number of machine learning algorithms, including the J48 decision tree classifier [26], support vector machines [12], and random forest algorithms [23]. Random forest classifiers, known to be one of the most accurate learning algorithms, combine multiple classification methods to achieve more predictive results. In particular, the random forest classifier builds a number of decision trees, where each node in a tree encodes a decision using one or more features that partition the input data. The leaves of each decision tree correspond to the set of possible labels (i.e., {`benign`, `malicious`}), and the output of all of the trees are then ensembled such that the average behavior among all trees is produced as the final decision. In our testing, the best ratio between detection rates (DT) and false positive rates (FP) were produced by the random forest classifier. Furthermore, the classifier is efficient enough to perform online detection in our application. Consequently, DISCLOSURE uses the random forest classifier to build its detection models.

We evaluated our detection models against NetFlow data collected from two networks: a university network (N_1) that does not apply sampling, and a large Tier 1 network (N_2) that samples one out of 10,000 flows. Figure 2 shows the detection rates (DT_1 for N_1 and DT_2 for N_2) and false positive rates (FP_1 for N_1 and FP_2 for N_2) for individual features sets, and all possible combinations among different feature sets. The feature sets we evaluated are the set of statistical features extracted from (i) the flow size (F_1); (ii) the flow size-based features extracted from the output of the autocorrelation function (F_2); (iii) unique flow sizes for each server (F_3); (iv) the combination of all flow size-based features (F_{all}); (v) the features for characterizing client access patterns (C_1); (vi) unmatched flow density (C_2); (vii) the combination of all client access pattern-based features (C_{all}); (viii) temporal features (T_{all}); (ix) the combination of client access pattern and flow size-based features ($F+C$); (x) the combination of flow size and temporal features ($F+T$); (xi) the combination of client access pattern and temporal features ($C+T$); and, finally, (xii) the combination of all feature sets ($F+C+T$).

Figure 2 indicates that individual feature sets are not as effective as combinations of multiple feature sets. Furthermore, increased levels of feature aggregation results in better detection rates with less false positives. Finally, we note that the most promising results were achieved on both data sets by using all possible feature sets as input to the classification process. Hence, DISCLOSURE uses detection models that include all features sets ($F+C+T$) to detect botnet C&C channels.

4. FALSE POSITIVE REDUCTION

NetFlow data, by its nature, provides limited information about the real activities that are carried out in a network. As a consequence, a botnet detection system based only on the analysis of NetFlow data could produce results that are likely to contain some false positives (FP).

As we explain in Section 5, DISCLOSURE can be tuned to decrease the overall FP rate to 0.5% or below. However, given the volume of NetFlow data that must be processed every day in large networks, even a misclassification rate less than a fraction of a percent can result in an unacceptably large number of false alarms. Note that some existing malicious activity detection systems have shown to be useful for specific classes of malware or attacks. Clearly, it would be beneficial to correlate the detection results of our system with the results of some previously built systems. Therefore, in our architecture, we include a component that has the aim of correlating the results that DISCLOSURE produces with the public feeds of other malware analysis or detection platforms. The main insight here is that by integrating different data sources, it is possible to further reduce DISCLOSURE's FP rate to a manageable level.

We have built a reputation-based component for FP reduction that uses three public services that provide reports about a wide

range of malicious activities on the Internet. The first service we make use of is FIRE [3,31]. FIRE is a system that identifies organizations and ISPs that have been observed to engage in malicious activities. FIRE's website reports detailed information about many autonomous systems (AS), including a maliciousness score, relative rankings among other ASes, as well as the number of C&C servers, exploit servers, and spam and phishing servers the AS has been hosting over time. In our implementation, we separate each type of information into two time series: one representing the current year, and one containing previous historical data. Afterward, we compute statistical features for each time series. For instance, for the time series built from the number of C&C servers observed before 2011, we compute the minimum, mean, and maximum values. After we repeat this step for each time series, we compute a final score by aggregating all the values together by assigning a weight of 0.8 to the value for the current year, and 0.2 to the previous years.

The second public service we use in our FP reduction component is EXPOSURE [2,5]. EXPOSURE is a system that uses passive DNS analysis methods to detect malicious domains. EXPOSURE currently analyzes data obtained from a large number of recursive DNS servers, and reports its findings on daily basis. For each domain, it provides the associated IP address list and the ASes in which they are located. Leveraging this information, we count the number of malicious domains detected in each AS and build a reputation score according to the density of maliciousness for each AS reported by EXPOSURE.

The last source of information we use for FP reduction is Google Safe Browsing [4], a service that reports maliciousness information about a large number of web sites. This tool can also be used to query specific AS numbers to obtain the percentage of web sites in that AS that host malicious services.

For each IP address that DISCLOSURE labels as a potential botnet C&C server, the FP reduction component fetches the associated AS number and corresponding reputation scores from FIRE, EXPOSURE, and Google Safe Browsing. Each of these individual reputation scores are then aggregated using a weighted linear combination. That is, given the reputation scores r_1, r_2, r_3 and corresponding weights w_1, w_2, w_3 for FIRE, EXPOSURE, and Google Safe Browsing such that $\sum_i w_i = 1$, the final reputation score R is calculated as

$$R = \sum_{i=1}^{3} w_i r_i,$$

where $0 \leq R \leq 1$. If R is below a tunable threshold we denote as RepThresh, this indicates that a particular server is located in a network that is historically *not* associated with malicious activities, and the corresponding alert is discarded as a FP.

We are aware of the fact that the FP reduction component can introduce an opportunity for the attacker to evade our system. For example, she could place her C&C server in a network with a high reputation score. However, note that this increases the burden on the attacker, and forces her to move away from more vulnerable targets located in ASes with lower reputation scores towards potentially better-protected networks. Therefore, we believe that, on a large scale, this is a favorable result.

5. EVALUATION

In this section, we present the design and results of several experiments we conducted to evaluate DISCLOSURE's detection accuracy, false positive (FP) rate, and performance. We also present deployment considerations, and conclude with a discussion of resilience to evasion.

The accuracy of DISCLOSURE's classification procedure greatly depends upon the environment in which the input NetFlows have been collected. For example, NetFlow collectors placed in a small company network versus those placed in a large ISP will likely observe significantly different volumes of traffic. To bound the storage requirements at each collector, sampling rates might be configured to match the particular traffic volume specific to each site.

Network	C&C Servers	Benign Servers
University Network (N_1)	892	1489
Tier 1 ISP (N_2)	2000	1742

Table 2: IP addresses in our labeled data set derived from data observed in N_1 and N_2.

To measure how DISCLOSURE responds to varying levels of sampling, we evaluated our system in two distinct environments: a medium-size network connecting multiple universities with no sampling, and a Tier 1 ISP network configured with a sampling rate of 1:10,000.

5.1 NetFlow Data Sets

Our NetFlow data sets were drawn from two separate environments: a university network located in Europe, and an ISP network located in the USA and Japan. Hereinafter, we refer to the university network as N_1 and to the Tier 1 ISP network as N_2.

Table 1 shows summary statistics for the two data sets. The N_1 data set was collected for a period of 18 days between the 7th and the 25th of September 2011. The NetFlow data of N_1 is not sampled and, therefore, all network flows present in the monitored network are represented in the data. The sensor in N_1 produced an average of 1.2 billion network flows per day. During this period, we collected 22 billion flows between 28 million unique IP addresses.

In contrast, we collected NetFlow data observed at N_2 for a period of 40 days between the 1st of June 2011 and the 10th of July 2011. The sensors in N_2 were configured to sample flows at a rate of 1:10,000. The data was harvested by 68 sensors, each of which was responsible for monitoring and forwarding NetFlow traffic collected from specific autonomous systems (ASs). The sensors collected approximately 400 million network flows per day between 50 million unique IP addresses.

5.2 Ground-Truth Data Sets

The accuracy of the classification models generated by a machine learning algorithm greatly depends on the quality of the training set [33]. In our case, to train the features used by DISCLOSURE, we required a ground-truth list containing both known C&C servers and known benign servers.

The malicious server data set consisted of 4295 IP addresses associated with real C&C servers observed in the wild during approximately three weeks preceding our experiments. The list of botnet C&C servers was provided to us by a company that specializes in threats intelligence. This list is used by the company as the core of their reputation-based detection engine. Hence, the aim is to be both complete with regard to current threats and keep FPs down. The reputation-based engine is deployed on a few hundred sites, and operational experience shows that FPs are minimal and coverage is at least comparable with deployed anti-virus tools (i.e., the engine captures threats that AV software installed on these sites misses).

We constructed our benign server training set from ranking information provided by Alexa [1]. In this case, we assume that the top 1,000 popular web sites reported by Alexa are not involved in malicious activities and, in particular, are not responsible for hosting botnet C&C servers. Alexa reports the top popular web sites grouped by geographical regions as well. In order to obtain a comprehensive list of benign servers, we combined the "Alexa Top 1000 Global Sites" with the most visited websites in the regions where N_1 and N_2 are located.

Once the benign domains lists were compiled, we resolved each DNS name on both lists to obtain the corresponding list of IP addresses. Note that we executed the DNS queries for each list from the same network geographical locations of the corresponding network (Europe for N_1, and US for N_2). Hence, the number of IP addresses collected for each network is different. This process resulted in 2,958 unique IP addresses for N_1, and 3,047 IP addresses for N_2.

Table 2 shows the number of benign and malicious servers in

Network	Sampling	Flows per day	Unique IP Addresses
Inter-University Network (N_1)	1:1	1.2 billion	28 million
Tier 1 ISP (N_2)	1:10.000	400 million	50 million

Table 1: Summary statistics for each of the two NetFlow data sets for N_1 and N_2.

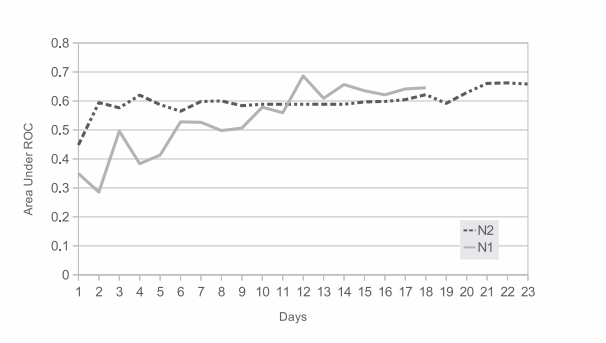


Figure 3: Area under ROC curves with different training set lengths for N_1 and N_2.

our labeled data set that were observed in the traffic of N_1 and N_2 respectively.

5.3 Labeled Data Set Detection and False Positive Rates

In the initial experiment, we evaluated DISCLOSURE's ability to recognize known botnet C&C servers from the ground truth constructed in the previous section. DISCLOSURE's detection rate (DR) and FP rates were measured by generating ROC curves for each data set under two configurations each that controlled the level of input data filtering performed prior to detection.

DISCLOSURE requires a minimum number of observed flows to a particular server in order to provide accurate results. This minimum is a threshold we denote by MinFlows, and can be set by a security administrator according to the volume of traffic at a particular site and any sampling that may be applied. We evaluated two values for MinFlows for each data set: 20 and 50. For each experiment, we excluded any servers that did not have at least one port that received more than MinFlows flows. We then evaluated the accuracy of DISCLOSURE's detection models by performing a 10-fold cross-validation.

We also considered varying the size of the training set as an additional tunable parameter. Figure 3 shows a summary of DISCLOSURE's accuracy, measured by computing the area under the ROC curve for different training windows. The curve for N_2 is almost constant. In comparison, the curve for N_1 steadily increases over the first 15 days before plateauing. This is due to the fact that the number of known C&C servers observed in the university network is low (see Table 2). Therefore, more time is required to collect enough data to properly train the models. For this reason, we decided to train DISCLOSURE with all the available data.

Figure 4 shows the individual ROC curves obtained by varying the classification threshold ClassThresh, i.e., the boundary separating benign scores from malicious scores, of DISCLOSURE's detection module. Consequently, each point in the ROC curves represents a possible setup configuration of the system. Security administrators can thus precisely tune DISCLOSURE to achieve a reasonable trade-off between FPs and false negatives based on the traffic characteristics of the network. Each graph also contains a short synopsis of possible working points. For example, configuring the system for a very high DR is usually too costly in terms of FPs. On the other end of the scale, it is often possible to achieve a 0% FP rate if we accept the fact that only one out of three C&C servers will be detected.

Point on the ROC curve	Servers Flagged as C&C (N_1)	Servers Flagged as C&C (N_2)
1.0% FP	12,383	4,937
0.5% FP	7,856	3,166
0.3% FP	6,295	1,958
0.0% FP	132	960

Table 3: Servers flagged as malicious by Disclosure for each of the networks N_1 and N_2 (without incorporating reputation scores).

Despite the differences between the two data sets, the results are similar. For instance, with MinFlows set to 20 flows and ClassThresh tuned to produce a 1% false positive rate, the system detects 64.3% of the C&C servers in the university network and 66.9% in the ISP network. This similarity emerges from the composition of all features, where the individual contribution of each feature is quite different in the two environments. For instance, most of the features are better suited to the unsampled data set, where traffic patterns are clearly preserved. However, some of the features—for instance, the *unmatched flow density*—provide the best results when applied to large networks, even in presence of a high sampling rate. The mixture of these two classes of features makes DISCLOSURE less sensitive to variability in the NetFlow collection environment and, therefore, more robust.

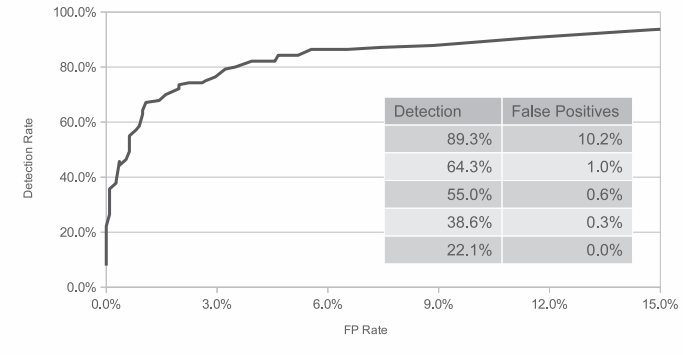
Another important difference between the two experiments is the fact that in the small network (N_1), DISCLOSURE provides better results for a higher value of the minimum flow threshold, while in the large network (N_2) it performs better with a lower threshold. This phenomenon is due to the fact that in the second case, the sensors are only collecting 1 flow out of every 10,000. Therefore, a high value for MinFlows would filter out all small-to-medium size botnets, leaving only a few large ones for the analysis. As a result, the features are now trained on a very few C&C samples and, therefore, tend to produce inaccurate models. This is an important issue to keep in mind when configuring the system. In general, if MinFlows is set too low, the features are exposed to samples that do not show sufficient regularity because an insufficient number of flows are observed in the traffic. If, on the other hand, MinFlows is set too high, the majority of the botnets are discarded, and the features are trained on too few samples. In both extremes, the result is a set of poorly trained models.

Finally, we manually verified the features of the benign servers that DISCLOSURE wrongly classified as being botnet C&C servers. In several cases, the network or the server were probably malfunctioning, and the clients (in most of the cases less than 10) were repeatedly trying to send the same data over and over again at regular intervals, and receiving no answer back from the server. This behavior, even though not malicious *per se*, is indeed quite similar to that exhibited by bot-infected machines.

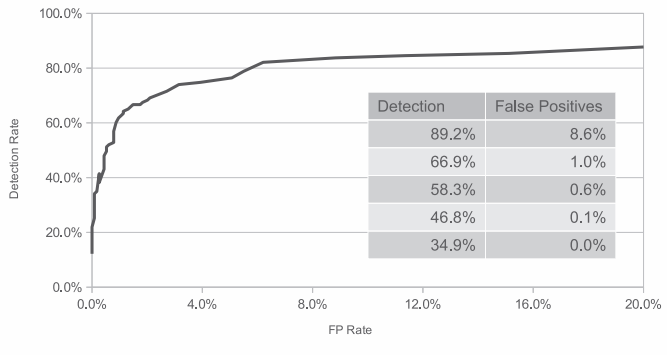
5.4 Real-Time Detection

In the previous section, we presented the results obtained with labeled data sets containing known benign and botnet C&C servers. In order to apply DISCLOSURE to the remaining unlabeled data, we needed to perform three separate operations.

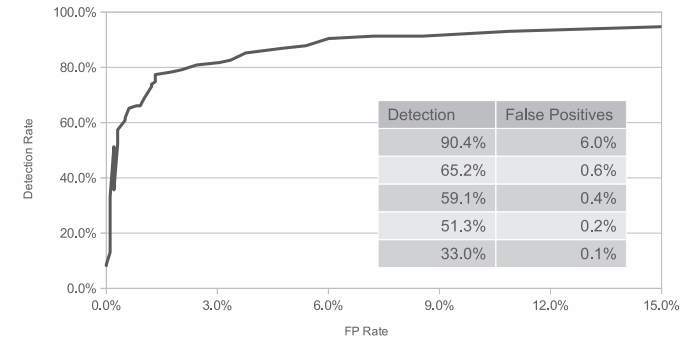
First, since DISCLOSURE is meant to discover C&C servers and not infected machines, we need to restrict the analysis to the servers only. In order to separate them from the clients, we apply the following heuristic: an IP address belongs to a server if the number of flows directed towards its top two ports (i.e., the two that receives the most connections) account for at least 90% of the flows



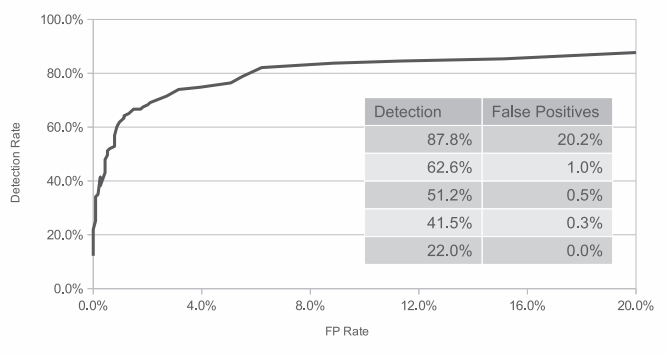
(a) N_1 with MinFlows $= 20$.



(b) N_2 with MinFlows $= 20$.



(c) N_1 with MinFlows $= 50$.



(d) N_2 with MinFlows $= 50$.

Figure 4: Classification accuracy for each data set (N_1 and N_2) with MinFlows $\in \{20, 50\}$.

Point on the ROC curve	C&C Servers after the RF (N_1)	C&C Servers after the RF (N_2)
1.0% FP	1779	1516
0.5% FP	1448	688
0.3% FP	1236	271
0.0% FP	20	91

Table 4: Servers flagged as malicious by Disclosure for each of the networks N_1 and N_2 (incorporating reputation scores). Here, RF refers to "reputation filter."

towards that address. From the count, we removed the ports used less than 3 times to filter out the noise generated by the fact that servers may also have outgoing connections. By adopting this technique, we identified 82,580 servers in N_1 and 530,011 servers in N_2.

The second step consisted of setting the value of the MinFlows threshold. According to the results obtained in the labeled data set, we decided to perform the rest of the experiments with the threshold set to 50 flows for N_1 and to 20 for N_2. After applying the threshold, we were left with 53,426 servers in N_1 and 48,713 in N_2.

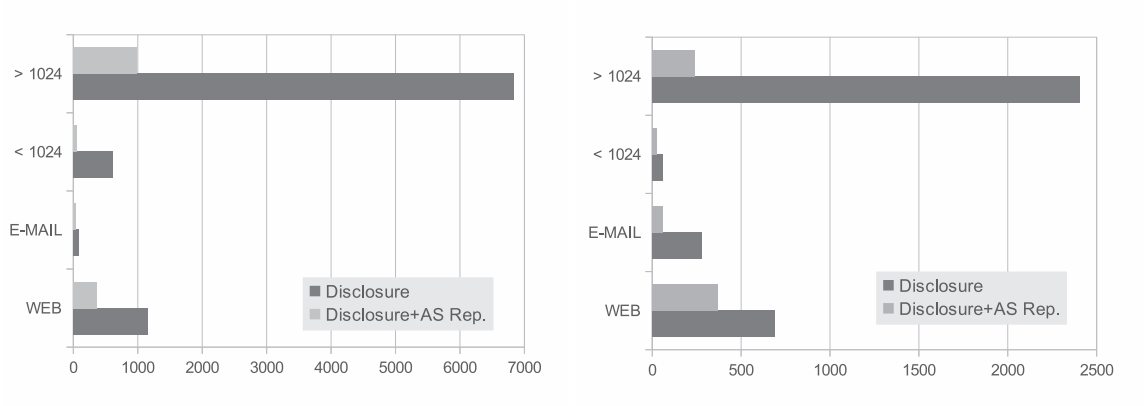
Finally, we needed to select the operational point on the ROC curve ClassThresh (i.e., the trade-off between DR and FP rates). Table 3 shows the number of servers detected in the two networks obtained with four different configurations of the system.

Despite the fact that the various configurations were chosen to minimize the number of FPs generated by the system, the number IP addresses suspected of being C&C servers is still relatively high. Therefore, to further reduce the probability of misclassification, we combined the results of Disclosure with a reputation score based on the information provided by EXPOSURE [2, 5], FIRE [3, 31], and Google Safe Browsing [4]. As explained in Section 4, this approach has the effect of narrowing down the results to the servers that have a higher probability of being malicious.

The way in which the reputation score is computed can be tuned according to the desired results and the number of daily alerts that the security administrator can tolerate. The more aggressive the filtering, the smaller the set of IP addresses flagged as C&C servers. In our experiments, we increased the strength of the FP reduction until we reduced the amount of alerts to a level that can be manually verified. The results are reported in Table 4.

Figure 5 shows the ports distributions of the C&C servers detected by Disclosure in the 0.5% false positive setting for N_1 and N_2. The graphs report the two most frequently used protocols: HTTP-related (ports 80, 443, 8080, 8000) and SMTP/IMAP (ports 25, 143, and 993). The remaining ports are grouped in two categories: the reserved ports (0-1023), and the registered and ephemeral ports (1024-65535). This classification is based only on the port number and not on identification of the true protocol. For instance, a botmaster can run a C&C server on port 25 to avoid firewalls, but that does not mean that he will adopt the SMTP protocol as well. It is interesting to note that the majority of the services identified by Disclosure run on ports higher than 1024. However, the distribution changes significantly after the FP reduction is applied. In fact, the reputation system filters out around half of the HTTP services, but cuts between 70 and 90% of the services running on high port numbers.

Finally, we manually investigated the C&C servers detected by Disclosure to gain some insight into the accuracy of the detection models and the reasons for misclassification. To this end, we chose the most conservative configuration: Disclosure configured for 0% FP + Reputation filter. With this setup, during one week



(a) N_1 port distribution.

(b) N_2 port distribution.

Figure 5: Port distributions of the C&C servers detected by Disclosure for both N_1 and N_2, with and without AS reputation scores.

of operation, DISCLOSURE reported 91 previously unknown C&C servers on the ISP network, and 20 on the university network.

We first manually queried popular search engines for each of the 111 entries. In 36 cases (32.4%), we found evidence of malware that was related to those IP addresses.[1] The fact that one third of our reports were confirmed by other sources is a strong support of the ability of DISCLOSURE to successfully detect C&C servers. Out of the remaining servers, 30 were associated with HTTP-related ports. After a manual investigation, seven of them seemed to be legitimate web sites—even though it is unusual that a small real estate company or a personal page in the Philippines would receive the large number of connections we observed in our traffic. Four pages were default placeholders obtained with a fresh installation of a web server; the number of NetFlow entries and varying flow sizes is suspicious, although this could be attributed to the web server not having a default virtual host configured. Four servers returned errors related to either unauthorized access or bad requests. Three of the HTTPS servers did not use the SSL/TLS protocol but some other form of binary protocol. The remaining servers were unaccessible at the time we checked them, which was approximately three weeks after the data was collected. Of the non-HTTP services, only four were still running at the time the checks were performed. Three of these appeared legitimate, but the remaining service was a web server located in Russia listening to a non-standard port. Finally, interestingly, eight servers were located in the Amazon cloud network, which is rapidly increasing in popularity for hosting ephemeral malicious services.

5.5 Performance Evaluation

As described in Section 2, the detection phase consists of two modules: feature extraction and detection. The detection module is highly efficient, requiring only several minutes to process an entire day's worth of data. Hence, detection performance is constrained by the analysis of input NetFlow data to extract the requisite features for analysis.

However, since the extraction of each feature is an independent process, the feature extraction procedure is an example of an embarrassingly parallel problem that can be easily distributed on multiple machines should the need arise. Nevertheless, even with the large amount of input data for our evaluation networks, we have not found it necessary to parallelize feature extraction. The current prototype implementation of DISCLOSURE consists of a number of Perl and Python scripts, all running on the same server: a 16 core Intel(R) Xeon(R) CPU E5630 @ 2.53 GHz with 24 GB of ram.

In the course of our experiments, we run all individual feature extraction modules in series in 10 hours 53 minutes for 24 hours of data. Therefore, DISCLOSURE is able to perform at approximately 2x real-time.

5.6 Deployment Considerations

To deploy DISCLOSURE to a real network, the administrator should configure three main settings: the minimum flows threshold MinFlows, the classification threshold ClassThresh, and the FP reduction threshold RepThresh. This setup can be accomplished by performing the following steps:

1. Choose the MinFlows threshold.
   This value should be selected according to the NetFlow sampling rate for the monitored network and the amount of available training data. If the threshold is set too high, the system will not have enough C&C samples to properly train. But, if it is set too low, the system will train on poor data, and produce inaccurate models.
2. Choose an operational point on the ROC curve for ClassThresh.
   This value should be selected according to the traffic volume of the network and the misclassification rate that can be tolerated. On one extreme, the system will be able to detect most of the C&C servers, but it will also generate too many FPs. On the other end of the scale, the system will miss many C&C servers, but the results will be much more precise.
3. (OPTIONAL) Apply and tune the FP reduction module using RepThresh.
   To reduce the number of alerts in large networks, DISCLOSURE can be coupled with other detection or verification techniques. In this paper, we propose the use of an AS reputation-based score to filter the servers hosted in benign networks. The weights for the constituent reputation systems can be modified to have a more aggressive or a more lightweight filtering contribution, and the overall reputation score filtering strength can be adjusted by setting RepThresh.

5.7 Evasion Resilience

The detection approach presented in this paper is predicated on the assumption that existing botnets often exhibit a regular, detectable pattern in their communication with the C&C server. However, we have not discussed how strong this requirement is and how difficult it might be for an attacker to perturb this regularity to avoid detection.

To answer this question, we designed two botnet families (hereinafter B_1 and B_2) that attempt to evade our system by inserting a random delay between consecutive connections and a random

[1]This evidence included reports from ThreatExpert, various sandbox malware analysis tools, MaliciousUrl.com, or the offensive IP database.

length padding in each flow. In our implementation, we employed two different randomization functions. The first randomization function produces uniformly distributed values on a fixed range. This is intended to model a botnet in which the programmer uses a random number generator to select a value from a fixed range. The second family adopts a more sophisticated approach and generates random numbers from a Poisson distribution. In this case, we model a more complex scenario in which the botmaster tries to mimic the flow inter-arrival times of benign services, which are known to be well-approximated as a Poisson process [22].

In our experiment, we generated 300 C&C servers for both B_1 and B_2. First, we randomly specified the size of each botnet and the duration of its activity. Afterward, we created synthetic NetFlow data for each server, using one of the aforementioned randomization functions to generate random flow sizes and intervals between consecutive flows.

Each botnet was created according to the following parameters:

Botnet lifetime	1 - 33 days
Number of bots	1,000 - 100,000
Flow sizes	4 - 3076 bytes
Delay between flows	1 min - 1 hr

The only significant difference between the two botnet families is that for B_1, the delay between consecutive flows between each bot and the the C&C server was a uniformly-distributed random value between 1 minute and 1 hour. For B_2, the delay was, instead, drawn from a Poisson distribution whose *mean* was randomly chosen in the 1 minute to 1 hour range. We decided to set 1 hour as an upper bound since, in order to maintain a reasonable flexibility and control over the botnet, a botmaster must be able to to send commands to the infected machines with a delay that is no longer than an hour or two.

Finally, we added the synthetically-generated NetFlows to our labeled data set and re-ran the classification evaluation using a 10-fold cross-validation. In both cases, DISCLOSURE was able to detect all the experimental C&C servers belonging to B_1 and B_2. In addition, the addition of these synthetic botnets to the training set had the side effect of actually increasing the overall detection rate. In other words, some of the real botnets that were not detected by DISCLOSURE in our normal experiments were detected after we added the synthetic data. This implies that our detection models were not properly trained to detect this kind of variability in the C&C channel behavior. However, by adding many new samples with a randomized behavior to supplement the training set, DISCLOSURE was able to subsequently detect real botnets that present similar access patterns.

6. RELATED WORK

In the last couple of years, much work has been done to investigate the topologies of botnets, understand how they operate, and create novel approaches to detect them. In this section, we analyze and discuss the state-of-the-art in network-based botnet detection, as well as the previous work on NetFlow-based anomaly detection.

6.1 Network-based Botnet Detection

Botnet-related research can be divided to two groups: work that focuses on the measuring botnets [11, 15, 20, 27] and work that focuses on detecting them [6, 16–19, 21, 29, 32].

A number of botnet detection systems perform horizontal correlation. While initial detection proposals relied on some protocol-specific knowledge about the C&C channel [19, 21], subsequent techniques overcame this shortcoming [17, 29]. The main limitation of systems that perform horizontal correlation is that they need to observe multiple bots of the same botnet to spot behavioral similarities. This is significant because as botnets decrease in size [11], it becomes more difficult to protect small networks, and a botmaster can deliberately split infected machines within the same network range into different botnets.

A second line of research explored vertical correlation to be able detect individual bot-infected machines. A number of systems focus on specific protocols such as IRC [6, 16, 32]. More advanced systems in this category provide generic solutions. For example, BotHunter [18] correlates the output of three IDS sensors and a scan detection engine to identify different phases in the lifecyle of bots. Wurzinger et al. [35] automatically generates detection models to identify single bot infected machines without any a priori knowledge on the C&C protocol. The problem with the approach, however, is that the system is only able to detect known instances of botnets.

6.2 Anomaly Detection Through NetFlow Analysis

To date, there has been a considerable amount of research on anomaly detection using NetFlow analysis. While some of the works proposed anomaly detection methods to detect specific kinds of malware such as worms [34] or spamming botnets [28], others tried to propose more general approaches to distinguish malicious traffic from benign traffic [8, 13, 30].

Wagner et al. [34] present an entropy-based approach to identify fast worms in real-time network traffic. Dewaele et al. [13] extract sub-traces from randomly chosen traffic traces, model them using Gamma laws, and identify the anomalous traces by tuning the deviations in the parameters of the models. Brauckhoff et al. [8] present a histogram-based anomaly detector that identifies anomalous flows by combining various information extracted from multiple histogram-based anomaly detectors. Sperotto et al. [30] analyzed the time series constructed from both flow and packet sizes, and tested them to find whether they were sufficient for detecting general intrusions.

Another line of research focuses on the analysis of the impact of sampling methods applied on NetFlow traffic. Mai et al. [25] analyze a set of sampling techniques experimented with two classes of anomalies. The results show that all types of sampling techniques introduce a significant bias on anomaly detection. Another work [9] as well studied the impact of packet sampling on anomaly detection metrics. Their analysis concludes that entropy-based anomaly detection systems are more resilient to packet sampling because the sampling still preserves the distributional structure.

6.3 Botnet Detection with NetFlow Analysis

Only a few papers exist that propose to use NetFlow analysis to specifically detect botnets. For example, Livadas et al. [24] propose a system that identifies the C&C traffic of IRC-based botnets by using machine learning-based classification methods.

Francois et al. [14] present instead a NetFlow-based method that uses the PageRank algorithm to detect peer-to-peer botnets. In their experiments, the authors created synthetic bot traces that simulate the NetFlow behavior of three P2P botnet families.

Both works succeeded in the identification of a specific type of botnet traffic, IRC in the first case and peer-to-peer in the second. DISCLOSURE, on the other hand, can successfully detect C&C servers without any prior knowledge about the internals of the C&C protocol. Moreover, our experiments shows how DISCLOSURE can be used to perform real-time detection on large datasets.

7. CONCLUSIONS

Botnets continue to be a significant problem on the Internet. Accordingly, a great deal of research has focused on methods for detecting and mitigating the effects of botnets. While the ideal data source for large-scale botnet detection does not currently exist, there is, however, an alternative data source that is widely available today: NetFlow data [10]. Though it is attractive due to its ubiquity, NetFlow data imposes several challenges for performing accurate botnet detection. In particular, packet payloads are not included, and the collected data might be sampled.

In this paper, we present DISCLOSURE, a large-scale, wide-area botnet detection system that incorporates a combination of novel techniques to overcome the challenges imposed by the use of NetFlow data. In particular, we identify several groups of features

that allow DISCLOSURE to reliably distinguish C&C channels from benign traffic using NetFlow records: (i) flow sizes, (ii) client access patterns, and (iii) temporal behavior. Our experiments demonstrate that these features are not only effective in detecting current C&C channels, but that these features are relatively robust against expected countermeasures future botnets might deploy against our system. Furthermore, our technique is oblivious to the specific structure of known botnet C&C protocols.

We provide an extensive evaluation of DISCLOSURE over two real-world networks: a university network spanning a small country where no NetFlow sampling occurred, and a Tier 1 ISP where NetFlow data was sampled at a rate of one out of every ten thousand flows. Our evaluation demonstrates that DISCLOSURE is able to perform real-time detection of botnet C&C channels over data sets on the order of billions of flows per day.

8. ACKNOWLEDGEMENTS

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement no 257007, the National Science Foundation (NSF) under grant CNS-1116777, and Secure Business Austria. Engin Kirda also thanks Sy and Laurie Sternberg for their generous support.

9. REFERENCES

[1] Alexa Web Information Company. `http://www.alexa.com/topsites/`, 2009.

[2] EXPOSURE: Exposing Malicious Domains. `http://exposure.iseclab.org/`, 2011.

[3] FIRE: FInding RoguE Networks. `http://www.maliciousnetworks.org/`, 2011.

[4] Google Safe Browsing. `http://www.google.com/safebrowsing/diagnostic?site=AS:as_number`, 2011.

[5] L. Bilge, E. Kirda, C. Kruegel, and M. Balduzzi. Exposure: Finding malicious domains using passive dns analysis. In *18th Annual Network and Distributed System Security Symposium (NDSS'11)*, 2011.

[6] J. Binkley and S. Singh. An Algorithm for Anomaly-based Botnet Detection. In *Usenix Steps to Reduce Unwanted Traffic on the Internet (SRUTI)*, 2006.

[7] G. E. P. Box, G. M. Jenkins, and G. Reinsel. Time Series Analysis: Forecasting and Control. In *3rd eddition Upper Saddle River, NJ: PrenticeâĂŞHall*, 1994.

[8] D. Brauckhoff, X. Dimitropoulos, A. Wagner, and K.Salamatian. Anomaly extraction in backbone networks using association rules. In *ACM Internet Measurement Conference (IMC'09)*, 2009.

[9] D. Brauckhoff, B. Tellenbach, A. Wagner, M. May, and A. Lakhina. Impact of packet sampling on anomaly detection metrics. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, IMC '06, 2006.

[10] B. Claise. Cisco systems netflow services export version 9, 2004.

[11] E. Cooke, F. Jahanian, and D. McPherson. The Zombie Roundup: Understanding, Detecting, and Disrupting Botnets. In *1st Workshop on Steps to Reducing Unwanted Traffic on the Internet*, pages 39–44, 2005.

[12] N. Cristianini and J. Shawe-Taylor. An introduction to support vector machines and other kernel-based learning methods. In *Cambridge University Press*, 2000.

[13] G. Dewaele, K. Fukuda, P. Borgnat, P. Abry, and K. Cho. Extracting hidden anomalies using sketch and non gaussian multiresolution statistical detection procedures. In *Proceedings of the 2007 workshop on Large scale attack defense (LSAD'07)*, 2007.

[14] J. Francois, S. Wang, R. State, and T. Engel. Bottrack: Tracking botnets using netflow and pagerank. In *IFIP Networking 2011*, 2011.

[15] F. Freiling, T. Holz, and G. Wicherski. Botnet Tracking: Exploring a Root-Cause Methodology to Prevent Distributed Denial-of-Service Attacks. In *10th European Symposium On Research In Computer Security*, 2005.

[16] J. Goebel and T. Holz. Rishi: Identify bot contaminated hosts by IRC nickname evaluation. In *Workshop on Hot Topics in Understanding Botnets*, 2007.

[17] G. Gu, R. Perdisci, J. Zhang, and W. Lee. BotMiner: Clustering Analysis of Network Traffic for Protocol- and Structure-Independent Botnet Detection. In *Usenix Security Symposium*, 2008.

[18] G. Gu, P. Porras, V. Yegneswaran, M. Fong, and W. Lee. BotHunter: Detecting Malware Infection Through IDS-Driven Dialog Correlation. In *16th Usenix Security Symposium*, 2007.

[19] G. Gu, J. Zhang, and W. Lee. BotSniffer: Detecting Botnet Command and Control Channels in Network Traffic. In *15th Annual Network and Distributed System Security Symposium (NDSS)*, 2008.

[20] J. John, A. Moshchuk, S. Gribble, and A. Krishnamurthy. Studying Spamming Botnets Using Botlab. In *6th Usenix Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[21] A. Karasaridis, B. Rexroad, and D. Hoeflin. Wide-scale Botnet Detection and Characterization. In *Usenix Workshop on Hot Topics in Understanding Botnets*, 2007.

[22] D. E. Knuth. Seminumerical algorithms. In *The Art of Computer Programming, Volume 2, Addison Wesley*, 1969.

[23] A. Liaw and M. Wiener. Classification and regression by randomforest. In *R News*, volume 2/3, page 18, 2002.

[24] C. Livadas, R. Walsh, D. Lapsley, and W. T. Strayer. Using machine learning techniques to identify botnet traffic. In *the 2nd IEEE LCN Workshop on Network Security (WoNS'2006)*, 2006.

[25] J. Mai, C.-N. Chuah, A. Sridharan, T. Ye, and H. Zang. Is sampled data sufficient for anomaly detection? In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, IMC '06, 2006.

[26] J. Quinlan. C4.5: Programs for machine learning. In *Morgan Kaufmann Publishers*, 1993.

[27] M. A. Rajab, J. Zarfoss, F. Monrose, and A. Terzis. A Multifaceted Approach to Understanding the Botnet Phenomenon. In *Internet Measurement Conference (IMC)*, 2006.

[28] A. Ramachandran and N. Feamster. Understanding the network-level behavior of spammers. In *SIGCOMM Comput. Commun.*, 2006.

[29] M. Reiter and T. Yen. Traffic aggregation for malware detection. In *DIMVA*, 2008.

[30] A. Sperotto, R. Sadre, and A. Pras. Anomaly characterization in flow-based traffic time series. In *Proceedings of the 8th IEEE international workshop on IP Operations and Management*, IPOM '08, pages 15–27, 2008.

[31] B. Stone-Gross, C. Kruegel, K. Almeroth, A. Moser, and E. Kirda. Fire: Finding rogue networks. In *2009 Annual Computer Security Applications Conference (ACSAC'09)*, 2009.

[32] W. Strayer, R. Walsh, C. Livadas, and D. Lapsley. Detecting Botnets with Tight Command and Control. In *31st IEEE Conference on Local Computer Networks (LCN)*, 2006.

[33] S. Theodoridis and K. Koutroumbas. *Pattern Recognition*. Academic Press, 2009.

[34] A. Wagner and B. Plattner. Entropy based worm and anomaly detection in fast ip networks. In *SIG SIDAR Graduierten-Workshop uber Reaktive Sicherheit (SPRING'06)*, 2006.

[35] P. Wurzinger, L. Bilge, T. Holz, J. Goebel, C. Kruegel, and E. Kirda. Automatically generating models for botnet detection. In *ESORICS 2009 : 14th European Symposium on Research in Computer Security*, 2009.

Security Economics – A Personal Perspective

Ross Anderson

University of Cambridge Computer Laboratory

ABSTRACT

This paper describes the origins of security economics. The birth of this thriving new discipline is sometimes credited to a talk I gave at ACSAC in December 2001, but the story is more complex. After sabbatical visits to Berkeley in 2001–2 to work with Hal Varian, we organised the first Workshop on the Economics of Information Security in June 2002. Since then the field has grown to encompass arguments over open versus proprietary systems, the econometrics of online crime, the behavioural economics of security and much else. It has started to have a significant impact on policy, with security-economics studies of cybercrime and infrastructure vulnerability being adopted as policy in the EU, while security economics PhDs have got influential jobs in the White House and elsewhere.

Keywords

information security, economics

1. EARLY DAYS

The ACSAC organisers have asked me to talk about the history of security economics. This subject is often considered to have been started by a paper I gave at ACSAC in December 2001, entitled "Why Information Security is Hard – An Economic Perspective" [2]. This paper had actually been put together from the new material on security economics which I'd written in my 'Security Engineering' book that first appeared in the summer of 2001 [3], and had already got its first public airing at SOSP as an invited talk in October of that year. Other people also contributed significantly to getting the subject off the ground. I've been digging through the old emails to refresh my memory of what happened.

I first got to know Hal Varian, then an economics professor at Michigan, by email when we served on a program committee together. He then moved to Berkeley and I arranged to meet him for dinner while I was at the IEEE Security and Privacy event in Oakland in May 2000. After dinner, he drove me back to the Claremont, and we still had so much to talk about that we sat there in his car for about an hour, missing most of the conference drinks reception.

Hal had been thinking about the various online payment systems that were competing then, and from which PayPal would later emerge as the victor. He understood the importance of liability assignment as a critical feature of the growth of credit cards and was thinking about how this insight might apply there. He'd come across my 1993 paper "Why Cryptosystems Fail" which studied ATM fraud and described some of the liability shifts in debit card payments [4]. I'd been wondering why US banks spent less on security than British banks, despite bearing more liability for card fraud. I recalled I'd mentioned this in a 1994 paper 'Liability and Computer Security' [5], and it still puzzled me. Hal suggested it was a classic case of moral hazard: UK banks were shielded by inappropriate liability laws, so they got lazy and careless, leading to an epidemic of fraud, and gave me a copy of a new book, 'Information Rules,' that he'd written with Carl Shapiro [34]. A few days later, he sent me a draft of a column he'd written for the New York Times discussing this [39], which talked of diffuse responsibility leading to DDoS attacks and suggested placing more liability on ISPs; he also pointed me at law-and-economics analysis of where liability should go [35].

Hal's book had a big impact on my thinking. That summer, I was busy completing my 'Security Engineering' book, and increasingly I found that it was the story about incentives that linked up the different case histories and made them speak to the systems engineering principles that I was trying to distill and explain. On 9 September I emailed Hal to say that having read it a couple of times, I was "getting more and more aware of security failures that occur because people don't understand the underlying network economics". I asked him to proofread the security-economics arguments in the draft chapters on e-commerce and assurance, which became sections 19.6, 19.7 and 23.2, as these were where I used sustained economic arguments. His feedback was very useful in polishing these. There were plenty bits and pieces of economics elsewhere in the book; for example chapter 22 incorporates John Adams' thinking on the risk thermostat and other examples or moral hazard, giving an early entry to behavioral ideas, while section 22.6 on 'Economic issues', summarises a variety of other arguments about incentives.

As I'd been hired to the Cambridge faculty in 1995, I was due a year's sabbatical in 2001–2, and following this lively and productive exchange I started talking to Hal about taking some of that sabbatical in Berkeley, at which he was

ACSAC 2012 Orlando, Florida USA

enthusiastic. He also sent me a copy of his undergraduate microeconomics textbook.

Although Hal was an important inspiration for the early work on security economics, he was not alone. Andrew Odlyzko had already remarked that the poor user-friendliness of both Microsoft software and the Internet is due to the fact that both Microsoft and the Internet achieved success by appealing to developers at least as much as to users [31]; extending this insight to security was obvious once I'd read Hal's book.

My book finally went off to the printers in January 2001. I'd extracted from Wiley, the publisher, an agreement that I could carve out the new material on security economics into a paper, and this became 'Why Information Security is Hard – An Economic Perspective' which appeared at ACSAC in December 2001 and got about 17 listeners. In fact, it had already appeared as an invited paper before then at the Symposium on Operating Systems Principles in Banff, Alberta in October 2011 where the audience was somewhat larger. The idea that platforms like Windows are insecure not because Microsoft engineers are careless, but because Microsoft had to appeal to developers at least as much as to end users in order to win platform races against the Mac and against OS/2, seemed to grab the SOSP crowd as a new and interesting angle. The paper also exposed them to the consequences of statistical failure models [14], where the black hats can attack anywhere but the white hats must defend everywhere; and the effects of asymmetric information in creating lemons markets [1].

2. FROM 9/11 TO THE FIRST WEIS

By the time I gave the SOSP talk at Banff, the 9/11 attacks had taken place, and security had suddenly become salient to a lot of people who hadn't given it much thought before. My own thoughts on the attacks were still evolving, but I put a few paragraphs on them in the ACSAC version of the paper just before the submission deadline. Like many people I was seriously alarmed that an enraged overreaction would do serious damage to the prospects for peace in the Middle East and for civil liberties, but I saw the invasion of Afghanistan as inevitable. I drew an analogy with piracy in the early 19th century, where the infant USA fought the first Barbary war with Algeria and Tunisia in 1801–5 because of attacks on US shipping, despite its aversion to colonialism and noted: "Liberals faced tough moral dilemmas: was it acceptable to conquer and colonise a particular territory, in order to suppress piracy and slavery there?"

I also remarked that "I believe that the kind of economic arguments advanced here will be found to apply to protecting bricks as much as clicks". As the war rhetoric ramped up, it was clear that no individual could push back on the public mood; the few academics who bravely tried to warn of the ever-less rational approach to risk (such as John Adams) were given a hard time in the media or just ignored. I reckoned that the best contribution I could make would be to build up the systematic approach. The academic method, like the proverbial mills of God, may grind slow but it grinds everything pretty small in the end. Our mission should be to understand risk, systems, crime and conflict; to build the models; to collect the data; and to be ready with solid policy advice once the world returned to its senses.

From Banff I flew to Berkeley and spent October-November 2001 there on sabbatical. Hal and I talked about security with other economists, such as Carl Shapiro and Suzanne Scotchmer, and laid plans to hold the first Workshop on the Economics of Information Security. I got acquainted with the work of George Akerlof, a Berkeley professor who'd just won the Nobel for his pioneering work on asymmetric information; his 'market for lemons' paper describes how used cars sell at a discount because buyers can't tell good ones from bad ones. This applies in spades to security products. Other influences included Jack Hirshleifer, the founder of conflict theory. By then search engines had arrived in the form of Altavista, so we were able to find a number of isolated early articles applying economic ideas to security, track down their authors and invite them.

Next stop was Singapore for two weeks at the end of November and start of December; while there I taught a crypto course and read a lot on environmental economics. I started to think about other aspects of lockin – for example, it cost Brazil a fortune to move its cars from petrol to alcohol – and about scaremongering, which was a big topic in the environmental debate at the time as well as the main response to 9/11 of many governments and security vendors. After Christmas at home, it was on the road again; I spent January–February 2002 at MIT, mostly working on technical topics such as API security and next-generation peer-to-peer systems; this led in due course to two large collaborative projects between Cambridge and MIT.

I went back to Berkeley for May-June 2002, staying from the Oakland conference through WEIS. We started to realise that with security economics we'd hit a sweet spot. The theoretical computer science crowd at Berkeley were working simultaneously on algorithmic mechanism design; the first paper may have been one of Hal's [37], but a recent (2001) paper by Nisan and Ronen [29] was inspiring security work such as the paper by Joan Feigenbaum and others on strategy-proof BGP routing [16]. At SIMS, John Chuang was also working on networks, mechanism design and security. (There would be some exchange with this community over the years with overlap in authorship between WEIS and conferences such as ACM EC and Gamesec.)

The high point of the trip was WEIS. Hal and I had not been the only people to talk of security and economics, but WEIS brought the threads together. What had previously been occasional scattered observations suddenly acquired critical mass and became a 'discipline'.

In addition to my early papers on ATM security, there was a paper of Hal's, "Economic Aspects of Personal Privacy", which in 1996 had described how markets for information could enhance welfare if they stop us being pestered by irrelevant ads [38]. Hal in turn cited a 1996 Comm ACM paper by Laudon, which discussed market failures in privacy and suggested a national information market by extending celebs' right to get compensation for commercial use of their images to ordinary mortals and their "data images") [26]. Carl Landwehr pointed us at US DoD concerns at security market failure dating from 1991 when an NRC report pointed this problem out [28].

New work at the first WEIS included Alessandro Acquisti introducing behavioural economics into the field. He applied it to the economics of privacy on which he'd worked with Hal as part of his thesis, showing that the Pareto-optimal outcomes envisaged in Hal's 1996 paper could be prevented by information asymmetry caused by uncontrolled information spread, user myopia, and the small size of the market

for privacy. On the practical side, Jean Camp proposed a market in vulnerabilities, which rapidly turned into reality. She'd first written on security economics in 2000 at ISW, where she proposed tradable 'pollution' permits for insecure systems. (Vulnerability markets took off rapidly, with two startups already in 2002.) On the theory side, Larry Gordon, Marty Loeb and Bill Lucyshyn applied the literature on trade associations to analyse the need for information sharing on security vulnerabilities within specific industries, an issue that had become salient with the US government promotion of ISACs. For Microsoft, Barb Fox introduced us to the economics of standards, describing how Netscape had abandoned SSL and that "dead dinosaurs get eaten".

3. EARLY GROWTH AND 'TC'

Barb's talk foreshadowed the next security-economics issue that engaged me as soon as I got back to Cambridge, namely Trusted Computing. This initiative, by Microsoft, Intel and others, proposed the addition to PC motherboards of a Trusted Platform Module – essentially a smartcard chip to monitor the boot process and provide secure key storage, plus modifications to the Windows operating system that would have supported a virtual secure coprocessor. Now that we had the beginnings of an understanding of lock-in and its interaction with security, it was natural to see this as an attempt to increase the lock-in of Windows and Office; it seemed optimised for this rather than for decreasing the likelihood that end users would get their PCs infected. I wrote the 'TCPA FAQ' in late June 2002 setting this out [6], with a fuller version in July.

This got very widely read and communicated some basic security-economics ideas to an extremely wide audience. I was invited to give a talk at the prestigious Software Economics confrence at Toulouse that November, and wrote up the Trusted Computing critique along with ideas originating in [14] on the dependability of open versus closed systems into a paper I gave there [7]. The key insight was that if bugs are distributed according to the ideal model, then the equipartition property ensures that in the long term a system that's opened to inspection will be as reliable as one that isn't; the mean time to failure will depend only on the total time spent testing. Thus, to argue that an open system, or a proprietary one, would be preferable in a given circunstance, you have to argue that the system in question deviates somehow from the ideal. And just as a market failure can justify a regulatory intervention, so a deliberate anticompetitive play can call for even more vigorous action from the trustbusters. This was heady stuff, and Microsoft's number three Craig Mundie flew out from Redmond in a private plane to debate the issue.

WEIS 2003 was at the University of Maryland, ably hosted by Larry and Marty, and thanks perhaps to the publicity generated by the Trusted Computing debate we had double the numbers of the previous year. We invited John Manferdelli, the Microsoft executive in charge of trusted computing, to give a keynote talk, while I talked about the competition policy aspects. Even the name of the best became an issue; I pointed out that a 'trusted computer' is, in US military parlance, one that can screw you, and suggested that Microsoft ought to have called their baby 'trustworthy computing'. (Meanwhile, Richard Stallman named the beast 'treacherous computing'.) The trusted/trustworthy/treacherous computing debate would eventually die down in 2005 when Microsoft shipped Vista without it (they simply couldn't make it work), but in the interim 'TC' got us plenty of airtime. The talks by John and me were complemented by papers on DRM (the main initial application of TC), and on the effects of technical lock-in on innovation.

Meanwhile the breadth of the workshop increased dramatically; WEIS 2003 had papers on most of the topics that would exercise the community over the years to come. Threads emerged on evaluating the costs and benefits of security mechanisms and postures; on incentives to share vulnerability information; amd on what makes it possible for security innovations to succeed in the marketplace. There were more talks on two of the threads from 2002, namely Marty and Larry's model of capital investment in security, and the behavioural analysis of privacy that Alessandro had kicked off. Wny causes the 'privacy gap' – the fact that people say they value privacy, but behave otherwise? This divergence between stated and revealed privacy preferences would eventually become a subject in its own right. But that was for later.

4. ECONOMETRICS

From then on, we started to see more and more papers, panels and debates on issues at the sharp end. The third conference, in 2004, was hosted in Minneapolis by Andrew Odlyzko, and kicked off with a debate on responsible disclosure of vulnerabilities. Eric Rescorla argued that typical modern systems have so many vulnerabilities that while disclosing them will cause the vendors to work harder to fix them, it will not improve the software quality enough to compensate for the costs of easier attacks on systems that aren't patched promptly. Rahul Telang countered that without disclosure, the vendors wouldn't patch at all, and we'd end up with more attacks as knowledge of exploits spread. This debate was related directly to the work I'd done on reliability modelling, but was now fundamentally an empirical question: what was the actual distribution of bugs in large software products?

I got a new research student, Andy Ozment, who started looking at bug reports and collected the data. By the next WEIS, at Harvard in 2005, he concluded that software was more like wine than milk, in that it did improve with age [32, 33]. This provided empirical support for the current practice of responsible vulnerability disclosure, for which Rahul and others also collected other evidence.

WEIS 2005 also continued the research threads in security metrics, investment models and DRM, while adding a new theme: cyber-insurance. If online risks were as high as the industry claimed, why had the insurance industry not started selling billions of dollars a year in cyber-risk insurance, as optimists had predicted in 2002? Was it because correlated risks from flash worms and other 'monoculture' effects made the business uneconomic, or were the risks perhaps being overstated? Was it too difficult to assign liability, or to establish claims? Rainer Böhme, who raised these issues, had no clean answer at the time[1].

These themes heralded the arrival of what we might call the econometrics of security. In the early days the Internet

[1]We've noticed since that security breach disclosure laws helped fix the problem: if a company that loses personal data on 5 million customers suddenly has to write to them, that's a big enough expense that the insurers start to take an interest.

had been restricted to academics, so there were no attacks. Security researchers had little choice but to think of everything that could possibly go wrong, and reasoned about security by invoking a Dolev-Yao opponent who could intercept and modify all messages. However, after the dotcom boom the bad guys got online too, and as it's too expensive to assume Dolev-Yao opponents everywhere, we need to defend against what real bad guys actually do. Once we realised this, we started to acquire the necessary access to data on crime and abuse and the statistical skills to make sense of it. Tyler Moore, Richard Clayton and I wrote a number of papers collecting and analysing data on various types of cyber-crime [11], We teamed up with others to write larger reports on what goes wrong online and what governments might reasonably do about it.

The first of these, 'Security Economics and the Internal Market', was written for ENISA (the European Network and Information Security Agency) and appeared in 2008 [8]. It surveyed the market failures that underlie online crime and came up with a number of recommendations, most notably a security breach disclosure law for Europe following the model of most US states. This was eventually adopted by the European Commission and has now appeared in articles 31 and 32 of the draft of the EU Data Protection Regulation.

The second, on the 'Resilience of the Internet Interconnection Ecosystem', was also commissioned by ENISA [21]. Its subject was the resilience of the Internet itself, and whether the incentives facing communications service providers were sufficient to provide the necessary security and redundancy against large-scale failures, or the sort of attacks that might be mounted by a nation state. A specific concern was whether the BGP security mechanisms under development would be capable of deployment, or whether ISPs would act selfishly and not bother to turn them on. A more strategic concern was that the Tier 1 autonomous systems who make up the core of the Internet were starting to undergo rapid consolidation; where previously the Internet had been kept up by the efforts of a score of firms, which were large but not so large as to be indispensible, price competition had led to takeovers which in turn meant that Level 3 was accounting for almost a third of transit traffic. (Other firms such as Google and Akamai also had such market share that their failure could do serious harm to the Internet.) We concluded that regulatory intervention was premature, but that governments might usefully start paying attention to the problems and sponsoring more research.

The third was kicked off in 2011 when Detica, a subsidiary of the arms company BAe, published a marketing brochure which claimed that cyber-crime cost the UK £27 billion a year, and even persuaded the Cabinet Office (the UK government department responsible among other things for for intelligence oversight) to put their name on it. This was greeted with widespread derision, whereupon the Ministry of DefenceâĂŹs chief scientific adviser, Sir Mark Welland, asked us whether we could come up with some more defensible numbers.

By then there was a significant research community to tap into; in short order, Richard, Tyler and I recruited Stefan Savage, who had done significant work on tracking fake Viagra marketing; Michel van Eeten, who'd investigated the variability of botnet infection between ISPs; Rainer Böhme, who had collected a lot of data on insurance, stock scams and vulnerability markets; Chris Barton, who'd worked for McAfee; and Michael Levi, an expert on white-collar crime. The report that we produced from pooling our insights taught us something new, that online crimes inflict very much greater indirect costs than traditional villainy does. For example, in 2010 the Rustock Botnet sent about a third of all the spam in the world; we knew from Stefan's analysis that it made its operators about $3.5m, and from other figures that fighting spam cost about $1bn globally (some scaremongers claimed that the true figure was two orders of magnitude greater than this). We concluded that for every dollar the Rustock guys took home, they inflicted a hundred dollars of cost on the rest of us. Yet many of these scams are done by a small number of gangs.

The conclusion was simple: we should be putting more effort into arresting the bad guys and locking them up. Of course, the firms who sell spam filtering 'solutions' don't see the world this way, and they have a lot of lobbying clout. As a result, only a small fraction of cyber-security expenditures do much good. For example, the UK government allocated an extra £650m to cyber-security from 2011-2015, of which the police got only £20m – a stingy £5m a year. But at least we now have the data, and can start to point the finger at the anti-virus industry as being part of the problem rather than part of the solution.

Other work on the econometrics of online wickedness included a survey paper that Tyler, Richard and I wrote for the Journal of Economic Perspectives [11]. This is still a work in progress; we have a large grant from the DHS between Cambridge, CMU, SMU and the NCFTA to work on the economics of cybercrime. However, we now have the feeling that we're at the end of the beginning, and the numbers are starting to add up.

5. THE BEHAVIOURAL REVOLUTION

A second major thread to emerge over successive WIS conferences was what security engineers can learn from behavioural economics. This subject sits at the boundary between economics and psychology, and its subject matter is the psychological heuristics and biases that cause people to make systematic errors in their market behaviour. An example is myopia: as I noted, Alessandro Acquisti had explained at WEIS 2002 how people's failure to anticipate quite predictable future harms can laed to failure of the market solutions proposed by Hal for privacy problems. Also at WEIS 2002, Paul Thompson had talked about cognitive hacking, a variant on the same theme.

A few months after the first WEIS, the Nobel prize in economics was won by Daniel Kahnemann, who with the late Amos Tversky had pioneered the whole heuristics-and-biases approach. This got people's attention and started to move behavioural economics towards the centre stage. We had a steady stream of behavioural papers at successive conferences, mostly from Berkeley students, until WEIS 2007 when the workshop was held at CMU and we had a keynote talk from George Loewenstein, a distinguished behavioural economist on the faculty there. By then Alessandro Acquisti had moved to CMU, and taken up a joint post between George's Department of Decision Sciences and the Heinz business school; as CMU also has a strong technical security research team under Virgil Gligor and the world's largest security usability research group under Lorrie Cranor, this created a real centre of excellence.

At WEIS 2007, Alessandro, Bruce Schneier and I decided

that it was time to launch a Workshop on Security and Human Behaviour to create the space for the behavioural economics of security to grow. While we were seeing several papers in each WEIS, and some more in SOUPS (Lorrie Cranor's Symposium on Usable Privacy and Security) both of these had plenty papers on their core topics. We also felt that to fully develop the potential of the field we needed to tackle not just the privacy conundrum and the problems of password usability, but much broader and deeper issues such as the nature of deception both online and offline, and the mechanisms underlying our overreaction to terrorism. Dave Clark at MIT volunteered the premises and the first SHB took place in June 2008. SHB attracts mainly security engineers and psychologists, but with a decent sprinkling of anthropologists, philosophers and even a couple of magicians. It has also gone from strength to strength, though unlike WEIS (which now has over 100 attendees) it is restricted to about 60 invited participants to keep it interactive.

6. HORIZONS

In addition to the threads on the econometrics and behavioural economics of security that I've expanded on above, there are many more things going on. At recent WEIS events, many papers have built on the established themes of patch management, vulnerability markets and other information sharing mechanisms, investment models, regulation, open versus closed systems, rights management and insurance. But ever more new topics have opened up. Cormac Herley and colleagues at Microsoft have written a series of microeconomics papers on the incentives facing players in the underground economy [17, 18, 22, 23, 24]. Ben Edelman wrote a beautiful paper about adverse selction in certification, in which he showed that privacy-certified websites were more likely to invade your privacy than others and that the top search ad was more than twice as likely to be malicious as the top free search result [15]. My students Shishir Nagaraja and Hyoungshick Kim had fun playing evolutionary games on networks between agents and the authorites, gaining interesting insights about optimal strategies for both law enforcement and insurgents [25, 27].

By WEIS 2009, privacy failures in social networks had become a hot topic [13]. In 2010, we were titillated to learn that most of the pay-per-install industry was now carried on the back of pornography, or perhaps on other parts of its anatomy [41], while we also had a serious paper from the Federal Reserve about card fraud [36]. 2011 brought us a keynote talk on neuroeconomics at the scientific end, while at the practical end we had a depressing survey of the certification malpractices of the million top websites: only 5.7% do things properly [40]. And finally, at WEIS 2012 we learned that US cities with competing hospital groups as opposed to monopolies have significantly poorer information security practices [20], while one of the first papers on the macro aspects showed that participation in e-crime across countries varied with English proficiency, labour market size and the level of corruption of the local government, but not with per-capita GDP [19].

In retrospect, the growth of security economics has come in three directions. First, there has been a broadening of its scope, so we can now find papers on 'security and X' for many subdisciplines X of economics as well as psychology and a number of neighbouring humanities subjects. The second has been a deepening of those mines found to be profitable, such as the behavioural aspects and the econometrics of wickedness. Where you find gold, you keep digging! The third has been 'security economics of Y' for a number of application areas Y, ranging from payment cards through electricity meters to online pornography. There is still quite some room for work in each of these three directions.

7. CONCLUSIONS

The study of security economics began seriously in 2000. It was given a huge boost by two factors. The first was 9/11 and the subsequent rapid growth of the security-industrial complex, of which security economists have been among the most consistent critics. The second was the increasing adaptation of technical security mechanisms to support restrictive business models, from the Trusted Computing initiative through various kinds of DRM. Security economics has shed light on many other interesting topics, from cooperation and conflict in networks, through the drivers for security technology adoption, to the limitations of insurance markets.

In this article I've presented only a small snapshot; the interested reader should look at our survey paper [9], and then at our security economics web page [12] for more. For an even bigger picture see The Oxford Handbook of the Digital Economy, which contains surveys not just on the economics of security and privacy, but also of many related fields such as teh regulation of the Internet, software platforms, card payments, auctions, reputation systems, price comparison, social networks and copyright infringement [30].

Over the last fifteen years or so, the world economy has been disrupted – and white-collar crime has been revolutionised – by the Internet. We're not finished yet by any means; as computers and communications become embedded invisible everywhere, ever more devices and services will start to have an online component. Ever more industries will become a bit like the software industry. As always, the utopians will be wrong: crimes and conflicts will not cease, but will increasingly have some online aspect too. The concepts and tools of security economics will in time become one of the many frameworks we all use to understand the world and our place in it.

8. REFERENCES

[1] George Akerlof, "The Market for 'Lemons': Quality Uncertainty and Market Mechanism," Quarterly Journal of Economics v 84 (August 1970) pp 488–500

[2] Ross Anderson, "Why Information Security is Hard – An Economic Perspective", in *Proceedings of the Seventeenth Computer Security Applications Conference* IEEE Computer Society Press (2001), ISBN 0-7695-1405-7, pp 358–365; also given as a distinguished lecture at the Symposium on Operating Systems Principles, Banff, Canada, October 2001

[3] Ross Anderson, *'Security Engineering – A Guide to Building Dependable Distributed Systems'*, Wiley (2001, 2008)

[4] Ross Anderson, "Why Cryptosystems Fail" in *Communications of the ACM* vol 37 no 11 (November 1994) pp 32–40

[5] Ross Anderson, "Liability and Computer Security– Nine Principles", in *Computer Security – ESORICS*

94, Springer LNCS vol 875 pp 231–245

[6] Ross Anderson, "TCPA / Palladium Frequently Asked Questions", v 0.2 (26 June 2002); "Trusted Computing Frequently Asked Questions" v 1.0 (July 2002); at `http://www.cl.cam.ac.uk/~rja14/tcpa-faq-1.0.html`

[7] Ross Anderson, "Open versus Closed Systems: The Dance of Boltzmann, Coase and Moore", in Open Source Software Economics 2002, at `http://idei.fr/activity.php?r=1898`

[8] Ross Anderson, Rainer Böhme, Richard Clayton and Tyler Moore, *'Security Economics and the Internal Market'*, ENISA, March 2008, at `http://www.enisa.europa.eu/pages/analys_barr_incent_for_nis_20080306.htm`; shortened version, "Security Economics and European Policy", appeared in *WEIS 08*

[9] Ross Anderson, Tyler Moore, "Information security: where computer science, economics and psychology meet" in *Philosophical Transactions of the Royal Society A* v 367 no 1898 (2009) pp 2717–2727

[10] Ross Anderson, Chris Barton, Rainer Böhme. Richard Clayton, Michel van Eeten, Michael Levi, Tyler Moore and Stefan Savage, "Measuring the Cost of Cybercrime", WEIS 2012

[11] "The Economics of Online Crime" (with Tyler Moore and Richard Clayton) in *Journal of Economic Perspectives* v 23 no 3 (2009) pp 3–20

[12] *Economics and Security Resource Page*, at `http://www.cl.cam.ac.uk/~rja14/econsec.html`

[13] Joseph Bonneau, Sören Preibusch, "The Privacy Jungle: On the Market for Data Protection in Social Networks," WEIS 2009

[14] Robert Brady, Ross Anderson and Robin Ball, *'Murphy's law, the fitness of evolving species, and the limits of software reliability'*, Cambridge University Computer Laboratory Technical Report no. 476 (1999)

[15] Benjamin Edelman, "Adverse Selection in Online 'Trust' Certifications,", WEIS 2006

[16] Joan Feigenbaum, Christos Papadimitriou, Rahul Sami, Scott Shenker, "A BGP-Based Mechanism for Lowest-Cost Routing", PODC 2002

[17] Dinei Florêncio, Cormac Herley, "Sex, Lies and Cyber-crime Surveys," WEIS 2011

[18] Dinei Florêncio, Cormac Herley, "Where Do All the Attacks Go?" WEIS 2011

[19] Vaibhav Garg, Chris Kanich, L Jean Camp, "Analysis of ecrime in Crowd-sourced Labor Markets: Mechanical Turk vs. Freelancer", at WEIS 2012

[20] Martin S Gaynor, Muhammad Zia Hydari, Rahul Telang, "Is Patient Data Better Protected in Competitive Healthcare Markets?", at WEIS 2012

[21] Chris Hall, Ross Anderson, Richard Clayton, Evangelos Ouzounis, and Panagiotis Trimintzios *'Resilience of the Internet Interconnection Ecosystem'*, ENISA, April 2011; abridged version published at WEIS 2011

[22] Cormac Herley, "The Plight of the Targeted Attacker in a World of Scale," WEIS 2010

[23] Cormac Herley, "Why do Nigerian Scammers say they are from Nigeria?" WEIS 2012

[24] Cormac Herley, Dinei Florencio, "Nobody Sells Gold for the Price of Silver: Dishonesty, Uncertainty and the Underground Economy," WEIS 2009

[25] Hyoungshick Kim, Ross Anderson, "An Experimental Evaluation of Robustness of Networks," in *IEEE Systems Journal – Special Issue on Security and Privacy in Complex Systems*, Mar 20 2012

[26] KC Laudon, "Markets and privacy", Communications of the ACM Vol 39 no. 9 p 104 (1996)

[27] Shishir Nagaraja, Ross Anderson, "The Topology of Covert Conflict," WEIS 2006

[28] National Research Council, *Computers at Risk*, System Security Study Committee, CSTB, National Academy Press, 1991. Chapter 6, "Why the Security Market Has Not Worked Well", pp.143-178. Available at www.nap.edu

[29] Noam Nisan and Amir Ronen, "Algorithmic Mechanism Design", in *Games and Economic Behaviour* Vol 35 (2001) pp 166–196

[30] Martin Peitz, Joel Waldfogel, *The Oxford Handbook of the Digital Economy*, OUP 2012

[31] Andrew Odlyzko, "Smart and stupid networks: Why the Internet is like Microsoft", ACM netWorker, Dec 1998, pp 38–46, at `http://www.acm.org/networker/issue/9805/ssnet.html`

[32] Andy Ozment, "The Likelihood of Vulnerability Rediscovery and the Social Utility of Vulnerability Hunting", WEIS 2005

[33] Andy Ozment, Stuart Schechter, "Milk or wine: does software security improve with age?", 15th USENIX Security Symposium (2006)

[34] Carl Shapiro, Hal Varian, *'Information Rules'*, Harvard Business School Press (1998), ISBN 0-87584-863-X

[35] Steven Shavell, *'Economic Analysis of Accident Law'* (Harvard 1987)

[36] Richard Sullivan, "The Changing Nature of US Card Payment Fraud: Issues for Industry and Public Policy", WEIS 2010

[37] Hal Varian, "Mechanism design for Computerised Agents" (1995)

[38] Hal Varian, "Economic Aspects of Personal Privacy" (1996)

[39] Hal Varian, "Managing Online Security Risks", Economic Science Column, The New York Times, June 1, 2000, `http://people.ischool.berkeley.edu/~hal/people/hal/NYTimes/2000-06-01.html`

[40] Nevena Vratonjic, Julien Freudiger, Vincent Bindschaedler, Jean-Pierre Hubaux, "The Inconvenient Truth about Web Certificates"

[41] Gilbert Wondracek, Thorsten Holz, Christian Platzer, Engin Kirda, Christopher Kruegel, "Is the Internet for Porn? An Insight Into the Online Adult Industry," WEIS 2010

Trust Engineering — Rejecting the Tyranny of the Weakest Link

Susan D. Alexander
Intelligence Advanced Research
Projects Activity (IARPA)
IARPA/MS2 Building/ODNI
Washington, DC 20511
(301)851-7505
Susan.D.Alexander@ugov.gov

ABSTRACT
In 2002 [1], the National Security Agency's Information Assurance Research Group coined the term, *trust engineering*, to describe a methodology for making use of software of uncertain provenance in mission-critical systems. Today, the loss of control that made software so hard to trust then applies to the rest of the supply chain as well. The discipline we described in the internal paper, Trust-engineering: An Assurance Strategy for Software-based Systems, no longer seems heretical today, even at NSA. Ten years later, we revisit the principles of trust engineering, compare the mechanisms available to us today with the practices of the past, and explore the construction of systems that are stronger than their weakest link.

Categories and Subject Descriptors
C.2.2 [**Computer Systems Organization**]: Computer Communication: Networks – *Security and Protection*.

General Terms
Design, Security, Theory.

Keywords
Trust engineering, assurance, isolation, monitoring, containment, trust, defense in depth, provenance, supply chain, composition, layered, moving target defense, virtualization.

1. INTRODUCTION
This paper contains no research results, few references, and no new hacks nor counter-hacks. It is an essay in the philological sense, that is, it has as its goal to *try out* a system of ideas on a new audience. The ideas in *this* essay had their origins at a time and within a culture that did not often publish its results outside of Government. Though, in 2002, our target audience was the NSA information assurance establishment, and we wrote our paper at the behest of the information assurance deputy director, the impetus was the inescapably increasing intersection of commercially-produced components with our business. There had been talk of a Commercial Off-The-Shelf (COTS) strategy for some time, but few really grasped the implications of such a shift. With trust engineering we sought both to introduce our Products organization to the inevitability that it would have to grapple with untrusted components in order to deliver the functionality for which its customers were clamoring, and to the possibility that there was a methodology that could allow them to do this without severely compromising trust. This essay seeks to expand the circle of practitioners conversant with these concepts, as in the future the bulk of security design will of needs come from the private sector.

ACSAC '12, December 3-7, 2012, Orlando, FL
ACM 978-1-4503-1312-4/12/12

2. THE WEAKEST LINK
Over the past ten years, as the market for information technology outside of national security has exploded, sources of untrusted technology are (literally) everywhere, and many high-assurance providers have left the game. At the same time, because of software's potential to become the unwitting delivery mechanism for network attacks, the number of things that we might wish we *could* trust has vastly increased. A number of factors have contributed to this dynamic. For one, computers have made it possible to do far more interesting things with information than merely communicate it, and the appetite for new functionality has become insatiable. As a corollary to that observation, protection of information is not the killer app for most customers of the newer functionalities, and information technology vendors, rather than information security vendors, have increasingly taken on the job of providing it so far as it aligns with their business objectives[1]. Third, a flatter world has produced a dynamic, global IT supply chain offering state-of-the-art functionality much more cheaply than it can be obtained from vetted providers.

Though an information technology security strategy that relies upon perfect control of every element of the ecology is no longer practicable, the idea that the security chain is only as strong as its weakest link still informs many policy and technology decisions. The US strategic response to globalization of the supply chain, as exemplified by the Defense Department's Trusted Foundry program [4] and the Comprehensive National Cyber Initiative's Supply Chain Risk Management Initiative [7], has chiefly consisted of schemes for holding onto component assurance as the pathway to trust. Given the complexity of modern hardware and software, it is virtually impossible to determine assurance through inspection. These activities, therefore, often make provenance a proxy for trust, focusing, for example, on ensuring continued availability of domestic production facilities and sharing threat analyses of foreign suppliers. At the more tactical level, most readers will have encountered, in their campus or workplace, IT

[1] Some of these vendors have become quite serious about information assurance; it is a goal of this essay to further that progress.

acquisition security regulations that keep the chain strong by allowing installation of approved software only (more on this later). Just as often, it's users who are tagged as the weakest link, inspiring well-meaning but largely ineffectual initiatives to improve their bad behavior, viz. National Cybersecurity Awareness Month, National Internet Safety Month, and even Wombat's endearing Anti-Phishing Phil.

Though it may seem like a leap to think of integrated circuits, applications software and people as members of the same class, it is the major contention of this essay that all of these entities look and behave much more like components than they do like self-contained, self-protecting systems, and we would be well-advised to stop thinking of them as if they were the latter. When my friendly neighborhood information systems security guy tells me that I can't download that incredibly useful Zendian automatic translation software package because it might contaminate the network, he is giving me insight into this dynamic. He is implying that he views that software as a component, but he wants it to behave like a product, and preferably like a last-century high-assurance COMSEC device. It's a component because, should I download it, it will become part of the system that is the network and may bring harm to it. It needs to behave like a product because, unlike the components of that old COMSEC device, it enjoys no protections by virtue of being a constituent of a larger system.

3. THE GOOD OLD DAYS

But wait, that's backwards: Why should we expect the *system* to protect its *components*? If we can't rely on each component to ensure the security of the system how are we going to obtain trust? In the good old days wasn't every link in the chain a strong one? Counter to intuition perhaps, systems in the good old days did not resemble chains so much as organisms, and, in the best designs, components compensated for and interacted with each other in ways that obviated the need for absolute trust in any one element. Pursuing further the notion that today's products are yesterday's components, it seems likely that we might learn something valuable by examining the role of the latter in the construction of yesterday's systems. Perhaps some design principles will emerge that we can apply to turn our weak links into integral parts of some organic whole.

Since the products of the good old days chiefly provided confidentiality services, a piece of key-generation circuitry (KG) makes a good candidate for study. The typical COMSEC product expected its encryption engine to take in information, or *plain text,* and output cipher. The engine generally employed a cryptographic algorithm to produce a pseudo-random sequence that it could use to disguise the plain text. Commonly, the sequence took the form of a binary stream that was XORed to a binary form of the plain text. The product expected the engine to be strong and reliable, but under the hood, the engine actually sub-divided strength and reliability into a larger set of constituent properties and distributed responsibilities for them among a number of components. To KG, the engine gave responsibility for the strength and correct implementation of the cryptographic mechanism. Rightly fearing, however, that any individual piece of circuitry might fail, it replicated three copies of KG and assigned another component to take a majority vote of the three outputs and add that answer to the plain text. It encased all the pieces of the architecture that touched plain text in a tamper-resistant coating and enforced a one-way path through that subsystem. And, just in case, it gave still another component the job of testing the text coming out of the black box to check that plain text wasn't being inadvertently emitted. While it's evident that, without the assurances of correctness that KG brought, this COMSEC product would have merited no trust whatsoever, KG could only be relied upon to contribute that core functionality in the context of the fail-safe mechanisms surrounding it. At this point, a light should be dawning: The designers didn't actually trust the components! Each component was good at some things, but was not trusted to do others.

The system examined above demonstrates three essential dimensions that trust engineering can manipulate to build trust: assurance, isolation and monitoring. To understand how they function, it is useful to think of each of these concepts in relation to the correct behavior that should be expected of something trustworthy. We will say that a design element adds *assurance* to the system if it increases the likelihood of correct behavior, that it contributes *isolation* if it limits the effects of improper behavior, and that it provides *monitoring* if it detects improper behavior. In the example above, the redundant copies of KG contributed assurance—the probability that two KGs will fault simultaneously is much smaller than that of a single KG error[2]. The one-way path through the tamperproof module provided an example of isolation—even if the key generation circuitry completely fails, plain text is sealed in the box. Last, statistical monitoring of the cipher attempting to exit the containment zone raised the alarm if it detected plain text—this gives the system an opportunity to respond, for example, it can trigger a maintenance cycle or cause the device to power down.

4. BACK TO THE FUTURE

Having extracted some useful design principles implicit in the composition of past trusted systems, it remains to seek analogs in today's environment, and investigate whether invoking the trust engineering paradigm can help us design so that the apparent deficiencies of our weak links don't matter. Since our goal was to learn something about components and system trust, the preceding section focused on a device that was specialized for trust. We found, interestingly, that even a system charged to deliver a security service consisted of only-partially-trusted components. Today's products resemble those components; like KG, they could contribute critical functionality were they embedded within a system that compensated for their weaknesses, and allowed them to play only to their strengths. Since the elements of trust engineering are defined in terms of behavior rather than security, we can generalize the insights about trust obtained from the encryptor example to the types of systems most common today. We will now explore examples of ways to operate in the three dimensions presented in the last section using commercially-available capabilities.

4.1 Isolation

When we wrote the 2002 paper we had recently discovered that virtualization, at that time a new[3] tool used mostly for saving space in servers, could allow us to create containment zones similar to the plaintext, or *red,* space in the encryptor example. Using virtualization and our own security-enhanced Linux, our group had invented an architecture called NetTop™ capable of simultaneously hosting multiple virtual machines that could act as

[2] Exactly how much smaller depends on the expected failure rate.

[3] Virtualization was not a new concept, but the way VMware repurposed and made it work for the 21st century IT environment was.

containers for software [2]. The VMs could function equally to confine untrusted and/or malicious activity, or to seal in and protect critical trusted code from a surrounding hostile environment. As its name suggested (it stood for **net**work on a desk**top**), NetTop™ provided an architectural framework for constructing any system whose components could be expressed as software. The NetTop™ *virtual hardware* framework makes it fairly simple to solve the IT acquisition security problem posed in section 2. Downloading the Zendian software into one of the virtual containers confines any effects its bad behavior might otherwise have had on the enterprise, while leaving the user free to exploit Zendia's unique functionality to translate even the most obscure slang.

It has taken a surprisingly long time for virtualization to catch on as a system security tool, especially in acquisition security, where true comfort still only comes from line-by-line evaluations of software. Others, though, have seen the attractiveness of this approach's ability to reduce the need to know everything about the software we use. Containerized browsers and email quarantiners use virtualization to create spaces in which to perform those functions that require direct contact with the Internet and act as the most common vectors for malware. Instrumented virtualization-based sandboxes provide a means to safely observe the behavior of unfamiliar software and learn something about its nature[4]. While NetTop™ had access only to heavyweight desktop virtualization techniques that required virtual images of the whole computer in order to containerize anything, the field of virtualization has since progressed. Application virtualization and, now, micro-virtualization suggest the possibility of eventually isolating and constraining arbitrarily small processes so that we can design with them as we would have with the component circuitry of the good old days. Mobile virtualization, now on offer in new smart phone designs, may present a way to deliver the trust-enhancing power of isolation to a potentially huge user population.

4.2 Monitoring

The evaluators assigned to assess the trustworthiness of NetTop™ seized immediately upon the worrisome amount of security-critical software in the system. Even if the operating system and hypervisor started out in a good state, they contended, they could become corrupted or coopted later in the life cycle. This line of questioning led us to consider incorporating additional monitoring to increase the trustworthiness of the system. Handily for the exposition of this essay, we found we could use monitoring in two very different ways that, together, illustrate the interplay among the three trust engineering dimensions. We could monitor to backstop isolation failures by inserting tripwires that detected unexpected behavior. We could also bolster assurance claims for our critical components by monitoring their fitness for duty over time. The decision to attempt to monitor software integrity spawned efforts in measurement and attestation that continue today.

In the network environment, where we must expect software to be under constant attack, how to gauge the continued trustworthiness of components proves to be one of the thorniest problems. Recall that in the encryptor example the red portion of the architecture sat within a tamper-resistant module. Tamper resistance relies on both isolation and monitoring for its strength; it creates a barrier that makes it harder for malicious agents to reach the material needing protection, but, if well-conceived, it also exposes attempts to breach it by evidencing physical changes that monitoring will detect. We call barriers having the latter property *tamper-evident*. Measurement and attestation have the potential to provide a tamper evidence construct for software. Initial measurement of the software establishes a *known good* baseline against which to compare future measurements. In simple cases, that measurement could take the form of a digitally-signed hash of the software, with the key for the cryptography stored in a trust anchor such as the Trusted Platform Module (TPM). Whenever the monitoring dimension required, an attestation service could re-hash the software and compare with the authoritative value. Measuring components whose state we expect to change over time, for example, the SELinux host environment within NetTop™, require more sophisticated techniques which we will not discuss here; there are good papers in the literature. Making use of these types of mechanisms to effect tamper evidence can allow the construction of more robust architectures that can guard against misplacing continued trust in compromised components.

4.3 Component Assurance

We saw above that monitoring and isolation can help us to use untrusted or partially-trusted software cautiously by restricting privileges and intercepting behavior. We saw also that we could use those dimensions to preserve the integrity of software that we had reason to trust. In the assurance dimension we will seek means to increase the intrinsic likelihood that we will obtain proper behavior from our system although we include components in which we do not have perfect confidence. Returning once more to the good old days, the key generation scheme in the encryption engine suggests redundancy as one mechanism for achieving this goal. The engine added the majority vote function because, though it trusted the correctness of the functionality in the KGs, it was not so sure about their reliability. Polling three copies of KG increased confidence that the key generation portion of the engine would perform properly.

In the foregoing, the engine trusted the correctness of KG because world-class cryptographers had designed its algorithm and cleared defense contractors had implemented it. Confronted with components from unfamiliar or less-trusted sources, we typically know little about the processes that produced them. Though we can empirically determine that they exhibit behavior that we are looking for *some* of the time, we do not have the evidence to assert that they will exhibit *only* that behavior *all* of the time. In that situation redundancy does not improve assurance sufficiently. NSA has launched a relatively new initiative called Commercial Solutions for Classified (CSfC) [5], which addresses this dilemma through a combination of redundancy and *diversity*. For critical parts of an architecture, CSfC builds assurance by layering independently-sourced components having redundant functionality "such that each mechanism is sufficient should another get compromised [and] such that the mechanisms are independent: vulnerability or compromise of one does not imply compromise of another" [6]. While the redundant functionality improves the probability of assurance in the same way that the multiple copies of KG did, the diversity of implementation builds confidence further by reducing the likelihood that the components will fail together, accidentally or on purpose[5].

[4] Of course we can also use the sandbox as the basis for a monitoring mechanism.

[5] Philosophically, this property of diversity is actually a form of isolation.

5. ADDITIONAL DESIGN GOALS

Through analysis of just a few examples, we have touched on many of the uses and means of construction of the trust engineering dimensions. Let's review by systematizing our observations. We learned that we can establish *isolation* through either a moat or a prison mechanism, that is, we can either keep bad elements out or lock them in. Isolation can help to prevent exposure, prevent contamination and prevent collusion. We needed two elements in order to build *monitoring*: a triggering condition and an out-of-band mechanism to sense it. We saw that monitoring can enable us to detect both changes in behavior and changes in state. We obtained *assurance* through verification (cryptanalysts unable to break our encryption scheme), through construction (trusted developers doing implementation in a formal way) and statistically (through redundancy). We saw an interesting relationship between assurance and monitoring; sometimes it is convenient for state to be a proxy for behavior, but when we know little about state, the reverse relationship is more apt to be of use.

This essay has concentrated up until now on trust, but to truly reject the tyranny of the weakest link, we will have to go farther and think more holistically about risk. The formula Risk=VCT, where V is vulnerabilities, C is consequences and T stands for threat, provides a useful way of laying out the options for mitigation. Risk exists to the extent that someone has the capability and desire to exploit weaknesses in our systems, and those weaknesses result in significant disruptions to our plans. We can bring down risk by decreasing the value of any of its factors. In the foregoing exposition we did not distinguish between the loss of trust that results from our own actions and the loss occasioned by adversary attack. We examined a number of mechanisms focused on vulnerabilities—anything targeting improving the assurance of our mechanisms falls in that category, and we made some progress towards tempering the direness of consequences by using isolation to contain malicious behavior. So far, though, trust engineering has done nothing to reduce threat. If we consider the adversary's calculus, maximizing gain while minimizing cost, two additional design options emerge for consideration.

Remote network attacks are in so many ways an adversary's dream come true. The attacker, from a safe distance and at his leisure, performs the reconnaissance needed to detect vulnerabilities and fashion effective exploits. Then, because the expense and difficulty of upgrading and reconfiguring IT infrastructures causes them to change slowly, he mounts his attacks at his chosen moment, fairly secure in the knowledge that they will work, and for a long time. To tackle the threat component of risk directly, we need to decrease gain and increase cost. One way to drive up the cost of attack is to ensure that the nth attack costs no less than the first attack. Good cryptography works this way; the algorithm is designed so that the specific knowledge needed to read one message is useless for any other. In the network, *non-persistence* creates a similar effect. If the attacker has to reestablish a presence every time he wishes to do something, he cannot amortize attack development and deployment costs. We can use virtualization, for example, to establish temporary environments that do not allow an attacker to gain a permanent foothold. Perceived gain will decrease if pre-planned attacks work less often. Most network attacks depend on specific software and known configurations. Anything we can do to increase the attacker's *uncertainty* that the IT environment is still the same as when he did reconnaissance will decrease his motivation to attack. The area that has come to be known as *moving target defense* has brought more attention to the threat dimension and is stimulating research in ways to place the attacker at a greater trust disadvantage [3].

6. CONCLUSION

While the above did not attempt to be exhaustive, with any luck it has piqued the reader's appetite to learn more, and perhaps to experiment with some of these concepts. This essay will have been successful if it induces some cross-pollination among researchers, technologists and computer security professionals. Just like other engineers and architects, aspiring security designers would benefit greatly from having a basic foundation in the techniques and methodology available to them before beginning to build; an excellent outcome would be if this essay encouraged more security educators to couch the technology-specific approaches they teach in terms of a more abstract general framework, or if section 5 inspired some formal course development in trust engineering. If we really wish to escape the tyranny of the weakest link, we will have to adopt new attitudes, policies and practices. We will also have to project demand for robust tools for constructing systems that protect products that were not designed to defend themselves. Fortunately, a starting point exists in a rich body of practice and a promising set of technologies.

Because its goal was to expose the main tenets of trust engineering to a wider audience, this essay generally omitted discussion of the vulnerabilities of specific technologies. Admittedly, new attacks and new mitigations continue to appear, but hopefully the concepts presented here embody principles that can continue to be useful to designers when those attacks, and perhaps even the technologies they targeted, have been forgotten.

7. ACKNOWLEDGEMENTS

This essay draws heavily on the ideas of my former colleagues Bob Meushaw, Grant Wagner and Brad Martin, who were key contributors to the 2002 effort. Bob kindly read this paper and his, as always, excellent suggestions improved it. Thanks are due also to Larry Wagoner, for recommending me as the 2012 Invited Essayist and to my employer, the Intelligence Advanced Research Projects Activity (IARPA), for allowing me to participate in ACSAC.

8. REFERENCES

[1] Alexander, S. and Meushaw, R. 2002. *Trust-engineering: An Assurance Strategy for Software-based Systems*. NSA unpublished paper.

[2] Meushaw, R. and Simard, D. 2008. *NetTop Eight Years Later*. The Next Wave. 27, 3 (April, 2008) pp 10-21.

[3] National Cyber Leap Year Summit Co-chairs' Report, 2009: http://www.cyber.st.dhs.gov/docs/National_Cyber_Leap_Year_Summit_2009_Co-Chairs_Report.pdf.

[4] http://www.nsa.gov/business/programs/tapo.shtml

[5] http://www.nsa.gov/ia/programs/csfc_program/index.shtml

[6] http://365.rsaconference.com/docs/DOC-3509

[7] http://csrc.nist.gov/groups/SMA/ispab/documents/minutes/2009-04/ispab_mswanson-nist_april2009.pdf

SensorSift: Balancing Sensor Data Privacy and Utility in Automated Face Understanding

Miro Enev*, Jaeyeon Jung**, Liefeng Bo*, Xiaofeng Ren*, and Tadayoshi Kohno*

*Department of Computer Science and Engineering, University of Washington
**Microsoft Research, Redmond WA

ABSTRACT

We introduce *SensorSift*, a new theoretical scheme for balancing utility and privacy in smart sensor applications. At the heart of our contribution is an algorithm which transforms raw sensor data into a 'sifted' representation which minimizes exposure of user defined *private* attributes while maximally exposing application-requested *public* attributes. We envision multiple applications using the same platform, and requesting access to public attributes explicitly *not* known at the time of the platform creation. Support for future-defined public attributes, while still preserving the defined privacy of the private attributes, is a central challenge that we tackle.

To evaluate our approach, we apply SensorSift to the PubFig dataset of celebrity face images, and study how well we can simultaneously hide and reveal various policy combinations of face attributes using machine classifiers.

We find that as long as the public and private attributes are not significantly correlated, it is possible to generate a sifting transformation which reduces private attribute inferences to random guessing while maximally retaining classifier accuracy of public attributes relative to raw data (average *PubLoss* = .053 and *PrivLoss* = .075, see Figure 4). In addition, our sifting transformations led to consistent classification performance when evaluated using a set of five modern machine learning methods (linear SVM, kNearest Neighbors, Random Forests, kernel SVM, and Neural Nets).

Categories and Subject Descriptors

K.4 [**Computers and Society**]: Public Policy Issues—*Privacy*; I.2 [**Artificial Intelligence**]: Vision and Scene Understanding—*Modeling and recovery of physical attributes*; I.5 [**Pattern Recognition**]: Models—*Statistical, Neural Nets*; G.1 [**Numerical Analysis**]: Optimization—*Least squares methods*

1. INTRODUCTION

The minimal costs of digital sensors, global connectivity, computer cycles, in addition to advances in machine learning algorithms, have made our world increasingly visible to intelligent computers. The synergy of sensing and AI has unlocked exciting new research horizons and led to qualitative improvements in human-computer interaction. However, alongside these positive developments, novel privacy threats are emerging as digital traces of our lives are harvested by 3rd parties with significant analytical resources. As a result, there is a growing tension between utility and privacy in emerging smart sensor ecosystems.

Table 1: Data sharing models in sensor applications (all terms defined in Section 3).

	Platform	Application	Tradeoffs
M1	sensor data	get features, classify, app. logic	Innovation++ Privacy-
M2	sensor data, get features	classify, app. logic	Innovation- Privacy++
S.Sift	sensor data, sift gen., verify	classify, app. logic	Innovation+ Privacy+

In the present paper we seek to provide a new direction for balancing privacy and utility in smart sensor applications. We are motivated towards this goal by the limitations in the current models of data access in smart sensing applications.

At present there are two conventional modes of data sharing in smart sensing applications and they are either risk carrying or arbitrarily stifling:

In the first mode, an application is given access to all of the raw data produced by a sensor (i.e., the Kinect); the application is then free to do feature extraction, classification, and run the logic that powers its functionality. Although this model of data sharing is great for innovation it leads to a sacrifice in privacy (Table 1, first row M1).

In the second mode, an application is given access to some restricted set of API calls defined by the platform (i.e., Apple's iOS) which restrict access to the raw data produced by a sensor; the application can still perform classification and run its logic, however it no longer has direct access to the data. The benefit of this approach is that privacy can be significantly increased, however innovation is significantly diminished (Table 1, second row M2). Given the limitations of these interaction modes, we seek to find a new model of sensor data access which balances application innovation and user privacy. To this end we develop an information processing scheme called SensorSift which allows users to specify their privacy goals by labeling attributes as private and subsequently enabling applications to use privacy-preserving data access functions called 'sifts' which reveal some non-sensitive (public) attributes from the data (Table 1, third row S.Sift).

Our tool is designed to be understandable and customizable by consumers while defending them from emerging privacy threats based on automated machine inferences. At the same time this tool enables applications access to non-private data in a flexible fash-

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

ion which supports developer innovation. Importantly, while the private attributes must be chosen from a supported list (to enable data protection assurances) the public attributes requested by applications do not need to be known in advance by the SensorSift platform and can be created to meet changing developer demands.

Rather than developing a specific system instance, in this paper we tackle the challenge of protecting sensitive data aspects while exposing non-sensitive aspects. We overcome this challenge by introducing a novel algorithm to balance utility and privacy in sensor data and propose how to embed it in an information processing scheme which could be applied as part of a multi-application trusted platform.

Towards Privacy and Flexibility in Sensor Systems. Suppose that an application running on a camera-enabled entertainment system (like the Kinect) wishes to determine Alice's gender to personalize her avatar's virtual appearance. Suppose also that Alice (the user) has specified that race information should not be available to applications. At present, Alice can either avoid using the application (and thus sacrifice utility) or choose to use the application and forfeit her ability to ensure privacy.

A natural solution to this tension would be to allow data access which is based on pre-defined public and private attributes. While workable for well-known attributes like race and gender, this approach limits innovation as developers are restricted to the predefined public attributes. Under the SensorSift scheme, applications can opt to use standard public and private attributes or can propose novel public attributes not known by the platform in advance (private attributes are still defined by the system in advance and exposed to users as options).

Returning to our example, on a SensorSift supporting platform Alice can specify race as a private attribute. The system would then transform the raw camera data samples to adhere to this policy by maximally removing race information while exposing application-desired attributes. These public attributes could be anything defined by the developers — including attributes not known to the platform designers; for simplicity of exposition, however, we'll use gender as the public attribute.

The transformed sensor data would only be made available to the application if the system successfully verifies (using an ensemble of state-of-the-art classifiers) that the sifted data cannot be used to recognize the private attribute significantly beyond random guessing. If the sift is verified, the target application would receive the transformed data which could then be post-processed to infer the gender value.

Concept Overview. Given a particular sensor context (e.g., optical/image data) and fixed set of data features (e.g., RGB pixel values) the information flow through our scheme is as follows: users define private attributes and applications define (request) public attributes; developers use provided tools to generate a candidate transformation (sift) which attempts to expose the [arbitrarily chosen] public attribute(s) but not the specified private aspects of the data; the user's system checks the quality of the proposed sift using an ensemble of classifiers; and lastly, if the verification is successful the application is allowed access to the transformed data.

Typically we expect that the SensorSift platform will ship with many valid sifts that cater to standard application demands. More importantly, however, we offer support for application-supplied sift transformations which would be verified by the platform either at installation time or when the user changes his or her privacy preferences. Once a particular sift has been invoked and successfully verified it will be applied to each sensor data release. In the case where an application is using a known policy (standard public and private attributes) the platform can automate classification and simply release an attribute label. Alternatively, if the application needs access to a novel public attribute it will need to independently classify the sifted data it receives.

Evaluation and Results. To evaluate our approach, we test how well we can control the exposure of facial attributes in the PubFig database of online celebrity photographs [11]. We leverage the face image attribute scheme of Kumar et al. to provide a quantitative vocabulary through which privacy policies can be defined over facial features [10]. We then choose a set of 90 policies composed using 10 facial attributes (e.g., male [gender], attractive woman, white [race], youth [age], smiling, frowning, no eyewear, obstructed forehead, no beard, outdoors) and show that it is possible to successfully create data 'sifts' which remove selective facial characteristics in a discriminating manner to produce high classification accuracy for public attributes and low accuracy for private attributes (average *PubLoss* = .053 and *PrivLoss* = .075, see Figure 4).

In addition, our sifting transformations lead to consistent classification performance when evaluated using a set of five modern machine learning methods (linear SVM, kNearest Neighbors, Random Forests, kernel SVM, and Feed Forward Neural Networks). As an extension we also show that our approach maintains privacy when applied to complex policies (multiple public and/or private attributes) as well as dynamic video (i.e., sequences of data releases).

To our knowledge our approach is the first solution to verifiably decompose face images into sensitive and non-sensitive features when evaluated against state of the art machine classifiers. In addition, our framework enables on-demand computation and evaluation of sifting functions so that privacy and utility balance can be created for currently unknown but desired attributes (thus supporting application innovation).

2. THREAT AND USAGE MODELS

Smart sensing applications have already been adopted in numerous life-improving sectors such as health, entertainment, and social engagement [2, 9]. Driven by the diminishing costs of digital sensors, growth of computational power, and algorithmic advances even richer sensing applications are on the horizon [1]. In most instances, smart sensor applications create rewarding experiences and assistive services, however, the gathered raw data also presents significant privacy risks given the amount of personal information which modern algorithms can infer about an individual. The potential consequences of these risks are not fully understood given the novelty of the enabling technologies. Nonetheless, we feel that it is critical to develop ways of managing the information exposure in smart sensor applications preemptively rather than reactively.

To mitigate potential privacy threats posed by automated reasoning applications we propose to employ automated defenses. At a high level our scheme is intended to enable a quantitatively verifiable trade off between privacy and utility in participatory smart application contexts. A full description of SensorSift is provided in Section 3, yet intuitively, our goal is to create a trusted clearinghouse for data which transforms raw sensor captures into a sifted (or sanitized) form to maximally fulfill the privacy and utility demands in policies composed of user selected private attributes and application requested public attributes.

We envision a model in which applications are untrustworthy but, in general, not colluding (we discuss collusion in Section 9). Applications might be malicious and explicitly targeting the exposure of private attributes; more likely, however, they are well-intentioned applications that fail to adequately protect the data that

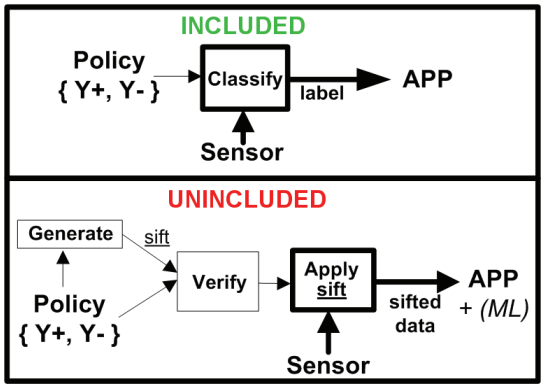


Figure 1: The two modes of operation in the SensorSift framework. Thick lines represent the processing elements that are occurring with each sensor release, while the thin lines indicate operations that are only necessary at application installation or policy revisions.

they harvest. We do not wish to expose private attributes to well-intentioned but possibly weak/insecure applications since those applications might accidentally expose the private information to other third parties. We also define as out of scope the protection of private attributes from adversaries with auxiliary information that might also compromise those private attributes. For example, we cannot protect the privacy of the gender attribute if an application asks for user gender during setup, and if the user supplies that information. We return to a discussion of these limitations in Section 9.

3. SYSTEM DESCRIPTION

While there are numerous potential deployment environments we primarily envision SensorSift running as a service on a trusted multi-application platform/system (like the Kinect) on which applications run locally.

Recall that the goal of the framework is to allow users to specify private attributes and allow applications to request non-private attributes of their choosing. At application install time (or when privacy settings are changed), the user and application declare their respective privacy and utility goals by creating a *policy* which contains the user desired private attribute(s) Y^- and application requested public attribute(s) Y^+. The user selected private attributes must always be known by the system to ensure that they can be verifiably protected; thus, the only viable *private attributes* are those for which the system's verification database has labels. Conversely, applications can request access to non-private (public) attributes which are unknown to the platform (i.e., developer invented). This makes it possible for the system to be one of two operating modes – included or unincluded policy mode.

In included mode (the simpler case), the user chosen private attribute(s) Y^- and the application requested public attribute(s) Y^+ compose a verified policy for which a data processing method is included in the platform. This means that the policy has been previously checked to ensure that the public attribute(s) do not leak information about the private attribute(s), and in addition, the platform has (shipped, or has been updated to include) a trained classification model which can recognize the public attribute(s) from the raw sensor data. As a result, it is possible to simply output the trained classifier's judgment on the public attribute to the application (as a text label) for each sensor sample request (Figure 1 top panel). From the application's perspective this is a straightforward way to get access to the public attribute(s) in the sensor data as all of the inference (pre-processing and classification) traditionally done by application logic is handled by the platform. We expect that many applications will opt to operate in this mode, especially if the list of platform included policies is large and frequently updated.

In some cases, the included list of attributes may not be sufficient to enable the application developers' functionality and utility goals. Whenever this is the case, the application interacts with the platform in unincluded policy mode (Figure 1 bottom panel). In this mode the user has selected some private attribute(s) Y^- (e.g., age) and the application is requesting access to some *novel* public attribute(s) Y^+ (e.g., imagine that eye color is a novel public attribute). Since support for this new policy is not included by the platform it is up to the application to provide a candidate sift (or data access function F) which can be applied to sensor data to balance the removal of private attribute information with the retention of application desired non-sensitive (public) data features. The proposed sift will only be allowed to operate on the sensor data if it can be successfully verified to not expose information about the private attribute(s) specified in the policy. While this scenario is more challenging from an application perspective, it is also more flexible and offers a way to meet the rapidly evolving demands of software developers.

Below we focus our discussion on the usage model for unincluded policies as it is unique to our approach and highlights all of the SensorSift framework's subcomponents.

Sift Generation. To create a candidate sifting function for an unincluded policy, applications can use our PPLS algorithm (defined in Section 4), develop their own method, or potentially use preverified sifts (e.g., crowd sourcing repositories). Code and documentation for the PPLS sifting generating function are freely available at `http://homes.cs.washington.edu/~miro/sensorSift` .

To use the PPLS algorithm, developers need to provide a dataset of sensor data (e.g., face images in our experiments) with binary labels for the public and private attributes. To facilitate the generation of this prerequisite labeled dataset, we imagine that developers will leverage freely available data repositories or use services such as Mechanical Turk.

Sift Verification. Once a candidate sift function has been provided to the platform, SensorSift must ensure that the proposed transformation function does not violate the user's privacy preferences. Indeed, there is no guarantee that a malicious application developer did not construct a sifting transformation function explicitly designed to violate a user's privacy. To verify that the transformation is privacy-preserving, SensorSift will invoke an ensemble of classifiers *ML* on the sifted outputs of an internal database *DB* to ensure that private attributes cannot be reliably inferred by the candidate sift. We discuss these components in more detail below.

Verification: Internal Dataset. The basis upon which we verify privacy assurances is a DB_{verify} dataset of sensor samples (i.e. face images) which would be distributed with each SensorSift install. For our purposes, we assume that the dataset is in matrix format X with n rows and d columns, where n is the number of unique samples (i.e., face images), and d is the dimensionality of each sample (i.e., face features). Large datasets with higher feature dimensionality offer attractive targets since they are more likely to capture real world diversity and produce stronger privacy assurances.

Verification: Classifier Ensemble. The second part of the verification process applies the candidate sift transformation to each sample in the internal database. Next an ensemble of machine classifiers are trained (using a training subset of the internal database) to recognize the private attributes with the sifted data. We leverage

state of the art methods which represent the most popular flavors of mathematical machinery available for classifcation including: a clustering classifier (**k-nearest neighbor** — parameters: q = 9, using euclidean distance metric with majority rule tie break; classifier source: MATLAB knnclassify), linear and non-linear hyperplane boundary classifiers (**linear-SVM** — soft margin penalty C = 10; classifier source: liblinear 1.8; **kernel-SVM** — soft margin penalty C = 10, radial basis kernel function, no shrinking heuristics; classifier source libsvm 3.1), a biologically inspired connection based non-linear classifier (**feedforward neural network** — 100 hidden layer neurons using a hyperbolic tangent sigmoid transfer function trained using gradient-descent backpropagation evaluated using mean squared normalized error, classifier source: MATLAB nnet package), and a recursive partitioning classifier (**random forest** — number of random trees per model = 500; classifier source: `http://code.google.com/p/randomforest-matlab/`).

For each *ML* model, independent training rounds are performed to obtain classifiers optimized for sifts of specific dimensions. A testing subset of the database is then used to evaluate how well the private attribute can be classified after it has been transformed by the proposed sift.

If any of the classifiers can detect the presence and absence of the private attribute(s) with rates significantly above the platform's safety threshold (e.g., 10% better than chance) the sift is rejected because it exposes private information. Alternatively if the private attribute accuracies on the sifted data (from the internal databse) are below the safety threshold the sift is deemed to be safe.

We again stress that while it is important for developers (or their applications) to evaluate the resulting accuracies on both public and private attributes, the system deploying SensorSift would in fact only verify that the private attribute classification accuracy is small.

Sift Application. If a sift has been proposed and successfully verified, it needs to be continuously applied with each data request made by the application. The application itself cannot apply the sifting transformation directly; this is requisite since, if the application had access to the raw sensor data it could be exfiltrated in violation of the privacy goals. Instead, the SensorSift applies the verified sifting transformations and outputs only the transformed data to the application.

Sift Post-Processing. In contrast to included mode where attribute labels are directly provided to the application, the application must post-process the sifted outputs (numerical vectors) that it receives in unincluded mode in order to determine the public attribute. This will likely involve running a classifier on the sifted sensor samples – the classifier can be trained using the database used to generate the sifts; once trained the classifier overhead should be minimal.

4. SIFT ALGORITHM - PPLS

In this work we create sifts using a novel extension of Partial Least Squares (PLS) that we call **Privacy Partial Least Squares**, or **PPLS**. At the heart of our technique is the long standing approach of using correlation as a surrogate for information. Given this perspective we design an objective function which simultaneously aims to maximize the correlations with public attributes and minimizes those with private attributes (while performing the structural projection of PLS). As we later show, this correlation-based PPLS algorithm is easy to use and also very effective within the context of automated face understanding; since our algorithm is domain independent we believe that PPLS is well suited to various datasets but this has not yet been verified.

Intuitively, our approach uses correlation between data features and attribute labels to find 'safe regions' in feature space which

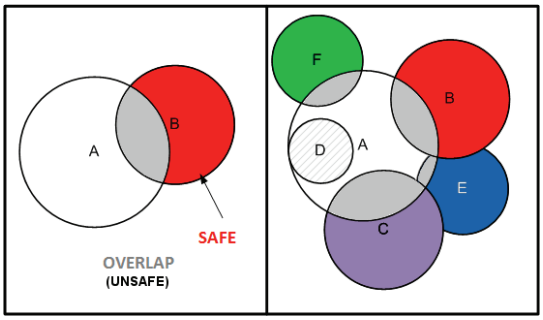


Figure 2: The left panel shows a simplified configuration of feature sets for two distinct attributes A (private) and B (public). The goal of SensorSift is to find the region(s) in feature space which are in the public feature set but not in the private one (i.e. indicated with the color red in the left panel). Raw data can be then re-represented in terms of how strongly it maps to this privacy aware region of the feature space. The right panel depicts how additional public attributes (C-F) which are invented by application developers map onto the feature space of our example. Note that in many cases it is possible to find privacy respecting regions of the region space through which to re-interpret (sift) raw data, however in some instances (attribute D) it may not be possible to separate attributes which have strong causal/correlation relationships (i.e. left eye color from right eye color).

are strongly representative of public but not private attributes (Figure 2). We then project raw sensor data onto these safe regions and call the result of the projection a loading or 'sift' vector.

Privacy Partial Least Squares Now that we have explored the intuition behind our approach we turn our attention to the details. To reiterate, Partial Least Squares (PLS) is a supervised technique for feature transform or dimension reduction [17]: given a set of observable variables (raw features) and predictor variables (attributes), PLS searches for a set of components (called latent vectors) that perform a *simultaneous* decomposition of the observable and predictor variables intended to maximize their mutual covariance. PLS is particularly suited to problem instances where the dimensionality of the observed variables is large compared to the number of predictor variables (this is generally true for rich sensor streams).

Let X be $[x_1,\cdots,x_{d_x}]^T$ a $n \times d_x$ matrix of observable variables (input features), and $Y = [y_1,\cdots,y_{d_y}]$ a $n \times d_y$ a matrix of predictor variables (attributes), where n is the number of training samples, d_x is the dimension of input features, d_y is the dimension/number of attributes. Without loss of generality, X, Y are assumed to be random variables with zero mean and unit variance. Any unit vector w specifies a projection direction and transforms a feature vector x to $w^T x$. In matrix notation, this transforms X to Xw.The sum of the covariances between the transformed features Xw and the attributes Y can be computed as

$$\mathrm{cov}(Xw,Y)^2 = w^\top X^\top Y Y^\top X w \tag{1}$$

The PLS algorithm computes the best projection w that maximizes the covariance:

$$\begin{aligned} find \quad & \max_w \left[\mathrm{cov}(Xw,Y)^2\right] \\ s.t. \quad & w\top w = 1 \end{aligned} \tag{2}$$

We propose a novel variant of PLS, *Privacy Partial Least Squares* (PPLS), that handles both public attributes and private attributes. Let $Y^+ = [y_1^+,\cdots,y_{d^+}^+]$ be a $n \times d^+$ public attribute matrix, and $Y^- = [y_1^-,\cdots,y_{d^-}^-]$ a $n \times d^+$ private attribute matrix, where d^+ is

Algorithm 1 Privacy Partial Least Squares

1. Set $j = 0$ and cross-product $S_j = X^\top Y^+$
2. if $j > 0$, $S_j = S_{j-1} - P(P^\top P)^{-1} P^\top S_{j-1}$
3. Compute the largest eigenvector w_j: $\left[S_j^\top S_j - X^\top Y^- (Y^-)^\top X\right] w_j = \lambda w_j$
4. Compute $p_j = \frac{X^\top X w_j}{w_j^\top X^\top X w_j}$
5. If $j = k$, stop; otherwise let $P = [p_0, \cdots, p_j]$ and $j = j+1$ and go back to step 2

the number of public attributes and d^- is the number of private attributes. We want to find a projection direction w that both maximizes the covariance $cov(Xw, Y^+)$ and minimizes $cov(Xw, Y^-)$.

This is achieved by optimizing the difference of covariances:

$$find \quad \max_w \left[cov(Xw, Y^+)^2 - \lambda * cov(Xw, Y^-)^2\right] \qquad (3)$$
$$s.t. \quad w^\top w = 1$$

The flow of the PPLS algorithm is outlined in the algorithm box (Algorithm 1). To transform X to more than one dimensions, we follow the PLS approach and develop a sequential scheme: we iteratively apply Equation 3, subtracting away covariances that are already captured in the existing dimensions (Step 2 in the Algorithm). Note that we only remove covariances from $cov(Xw, Y^+)$ but not $cov(Xw, Y^-)$, to ensure that every included dimension w is privacy-perserving by itself for all private attributes.

Free Parameters. There are two key free parameters of the PPLS algorithm, a λ term (privacy emphasis) and the number of sift dimensions to release K. In general we only release several dimensions from these sift vectors as a type of dimensionality reduction step which minimizes the risk of reconstruction. Despite the small size of the outputs sifts we find that public attributes can be correctly inferred with minimal accuracy degradation 7.

The λ term in Equation 3, represents the relative importance of privacy with higher λ values indicating an increased emphasis on removing private features (with a possible loss to utility).

5. DATASET

Our evaluation is based on the the Public Figures Face Database (PubFig) [11] which is a set of 58,797 images of 200 people (primarily Hollywood celebrities) made available as a list of URLs (see `http://www.cs.columbia.edu/CAVE/databases/pubfig/download`). The PubFig images are taken in uncontrolled situations with non-cooperative subjects and as a result there is significant variation in pose, lighting, expression, scene, camera, imaging conditions and parameters. Due to the size and real-world variation in the PubFig dataset we felt that it presents an appropriate foundation on which to evaluate SensorSift.

Validation, Alignment, and Rescaling. We began by downloading the PubFig image URLs using an automated script which would keep track of broken links and corrupted images. At the time of our data collection we found 45,135 valid URLS (77% of the advertised 58,797 images). For each image in the database PubFig provides four pixel coordinates which define the face region; we extracted this face region for each image aligned it to front-center (via affine rotation using the roll parameter). Next we rescaled each image to 128x128 pixels using bicubic interpolation and antialiasing.

Feature Extraction and Normalization. In addition to the raw RGB pixels, we extracted image derivatives of each face image to enrich the feature space of the raw data and provide a larger starting dimensionality to our algorithm. The four features we computed are popular in the computer vision literature and include raw RGB, image intensity, edge orientation, and edge magnitude [12]. After computing these transforms, we apply an energy normalization $(x-\mu)/(2\cdot\sigma)$ to the feature values of each face to remove outliers. Lastly, we concatenate all of the normalized image features for into a row vector and create a matrix to hold the entire dataset (45,135 rows/faces and 98304 columns/features per face).

PCA Compression. Next, we compute a PCA compression which is applied to the entire database (10:1 compaction ratio, $> 95\%$ energy maintained) to decrease the feature dimensionality of our face database and enable the PPLS algorithm to operate within reasonable memory constraints (16GB per node).

6. EXPERIMENTS AND METRICS

The privacy sifts that we compute are intended to provide quantitative assurances which adhere to a specified policy. Policies in turn are based on a set of user declared private attributes and developer requested public attributes. In this section we describe how we selected the attributes to include in the polices we evaluate. In addition we describe the metrics used to evaluate the quality of the sift generated for a particular policy.

6.1 Attribute Selection

The authors of the PubFig database were interested in providing a large vocabulary of attributes over each image to power a text-based 'face search engine' [10] Thus in addition to face coordinates and rotation parameters, each image in the PubFig dataset is annotated with classification scores for 74 different attributes. These scores are numerical judgments produced by a set of machine classifiers each trained for a unique attribute.

For analytical tractability we were interested in reducing the set of 74 available attributes to a more manageable number. Since we are using correlation as a proxy for information in our PPLS algorithm we analyzed the correlations between the available attributes to get a sense for the redundancy in the data.

We found two large clusters of attributes which were centered around *'Male'* and *'Attractive Female'*. The 'Male' attribute was very closely correlated with the attributes: *'Sideburns'*, *'5 oClock Shadow'*, *'Bushy Eyebrows'*, *'Goatee'*, *'Mustache'*, *'Square Face'*, *'Receding Hairline'*, and *'Middle Aged'*. Conversely, *'Attractive Female'* was very closely related to: *'Wearing Lipstick'*, *'Heavy Makeup'*, *'Wearing Necklace'*, *'Wearing Earrings'*, *'No Beard'*, and *'Youth'*.

Given their strong connection to a large set of the available attributes the *'Male'* and *'Attractive Female'* attributes were clear choices for our analysis, however we wanted to also get coverage over other characteristics which might be interesting from a privacy perspective. To this end we chose race (*'White'*), age (*'Youth'*), and emotional indicators(*'Smiling'*, *'Frowning'*), as well as other attributes which were descriptive about distinct regions of the face (*'No Eyewear,' 'Obstructed Forehead,' 'No Beard'*). Lastly we chose the 'Outdoors' attribute as it provides environmental context and it brings our total up to 10.

Policies. Having chosen a base set of 10 attributes we set out to evaluate how different choices of public and private attributes

would impact our goal of balancing utility with privacy. To this end we created 90 simple policies composed of all possible combinations of a single public and a single private attribute (e.g., public:*'Male'*, private:*'Smiling'*)[1].

6.2 Defining Mask Performance: PubLoss and PrivLoss

As previously stated, our system aims to produce data transformations which provide a favorable balance between utility and privacy given a policy instance P, dataset X, and attribute labels Y. Building on these concepts, we now introduce the quantitative measurements *PubLoss* and *PrivLoss* which judge the utility and privacy [respectively] achieved for sifts of a specified dimension within a given policy. *PubLoss* is intended to measure how much classification accuracy is sacrificed when public attributes are sifted (relative to their raw, unsifted versions), while *PrivLoss* is the difference between the highest classification rate of sifted private attributes relative to blind guessing.

- **PubLoss:** Decrease in F sifted public attribute classification accuracy relative to the achievable accuracy using raw (unsifted) data, computed as:

$$PubLoss = ML_m(X, Y^+) - ML_m(F_{Y^+,Y^-}(X,K), Y^+)$$

- **PrivLoss:** F sifted private attribute classification accuracy relative to chance, computed as:

$$PrivLoss = ML_m(F_{Y^+,Y^-}(X,K), Y^-) - .5$$

Where $ML_m(X,Y)$ denotes the Class Avg. Accuracy (Section 6.3) computed via classifier m using a 50%-50% split of training vs testing instances given data samples X with ground truth labels Y; and, $F_{c,d}(X,K)$ indicates the K dimensional privacy sift computed using data samples X and public and private labels Y^+ and Y^-.

A poor quality F would yield transformed samples whose public attributes are unintelligible and whose private attributes are easily identified (high *PubLoss* and *PrivLoss*). Conversely, an ideal sifting transformation would have no impact on the raw classification rates of public attributes while completely obscuring private attributes (no *PubLoss* and *PrivLoss*).

6.3 Classification Measures

The performance criteria we have selected (*PubLoss* and *PrivLoss*) are heavily dependent on measures of classification accuracy. Thus to provide stronger privacy claims, we now describe a robust approach to computing classification accuracy.

Class Average Accuracy. A common method of reporting classification accuracy is based on the notion of *aggregate accuracy* shown in Eq (4). Although this metric is suitable to many problem instances, whenever attributes have unequal distributions of positive vs negative samples (e.g., 78% of faces in our dataset lack eyewear) classifiers can achieve high *aggregate accuracy* scores by exploiting the underlying statistics (and always guessing 'no eyewear') rather than learning a decision boundary from training data. To avoid scores which mask poor classifier performance and warp our *PubLoss* and *PrivLoss* measures we opt to use **Class Avg. Accuracy** which is a more revealing gauge of classification success and is calculated as in Eq (5):

[1]We did not consider policies where the same attribute is both public and private

Table 2: Achievable accuracy for each attribute using raw data features computed using the maximum classification score across our five classifiers. Columns one and two use the *aggregate accuracy* metric and respectively represent our attribute recognition scores and state of the art performance (ICCV09 accuracies are reported from [11]). The remaining column provides the Class Average Accuracy measure.

Attribute	ICCV09	Agg. Accuracy	Class Avg. Accuracy
Male	81.22	94.18	92.86
Attr. Female	81.13	87.33	84.26
White	91.48	88.07	86.97
Youth	85.79	83.27	79.97
Smiling	95.33	92.11	87.69
Frowning	95.47	89.98	85.35
No Eywear	93.55	87.01	82.86
Obst. Forehead	79.11	81.01	77.86
No Beard	89.53	88.60	86.13
Outdoor	–	88.18	84.83

$$AggregateAccuracy = (tP + tN)/tS \quad (4)$$

$$ClassAvgAccuracy = (tP/(tP + fP) + tN/(tN + fN))/2 \quad (5)$$

Where tP is the number of True Positives (correct identifications), fP is the number of False Positives (type 1 errors), tN is the number of True Negatives samples (correct identifications), fN is the number of False Positive samples (type 2 errors), and tS is the number of Total samples ($tP + fP + tN + fN$).

As can be seen from equation (5) above, **Class Avg. Accuracy** places equal weight on correctly identifying attribute presence (positive hit rate) and attribute absence (negative hit rate) which in turn emphasizes classifier precision and offers less sensitivity to attributes with imbalanced ratios of positive to negative data.

6.4 Achievable Accuracies

Achievable Accuracy is a term we use to refer to the correct classification rates that we were able to obtain using the PubFig dataset. As mentioned in Section 6.1, images in the PubFig dataset are annotated with 74 numerical judgments produced by a set of 74 machine classifiers each trained to recognize a unique attribute. These scores are positive whenever the classifier has determined that an attribute is present and negative if the attribute is deemed to be absent (higher absolute values indicate additional confidence)[2]. To produce these numbers each attribute classifier was trained using 2000 hand labeled (ground truth) samples produced using Mechanical Turk [11]. Unfortunately due to the liability policy of Columbia University these ground truth labels cannot be released, instead we treat the classifier outputs as a proxy ground truth.

In the first two columns of Table 2, we use the *aggregate accuracy* metric to compare attribute recognition performance of our classifiers against state of the art methods. The third column provides the more robust **Class Average Accuracy** measure which we'll be using as the basis for result discussions. Note that all of the results in Table 2 are computed raw [unsifted] data features.

In the first column of Table 2 we report the correct classification rates of our 10 attributes from the original PubFig publication. These scores are based on the notion of *aggregate accuracy* shown in in Eq (4). In the second column of Table 2 we also use the *aggregate accuracy* method however we now apply classification

[2]Each scores indicates the distance of a sample from the SVM separation hyperplane

models which we train using the features described in Section 5. This serves as a verification that we are able to match state of the art results (in fact outperform for the first two attributes). In the last column we report the more robust classification measure - class average accuracy - which we use as a reference for the PubLoss computations for the remainder of the paper.

When looking at these accuracy rates it is important to note that the results could be improved with additional data, access to ground truth labels, and novel computer vision features. However we are not seeking maximal identification accuracy; instead the achievable accuracy serves as a reference point, and we are interested in how our sifting methods operate around it.

7. RESULTS

Below we describe the results of our experiments on the PubFig dataset. First we set a conservative privacy threshold and determine the sift output dimensionality that meets this criteria when measured against our ensemble of classifiers. Next we look at the *PubLoss* and *PrivLoss* computed from the 90 policies using one public and one private attribute, and describe the factors influencing the results. We follow this with an extension of our algorithm suited to complex policies (multiple public and/or private attributes). We also discuss how our approach can be applied to sequential sensor samples (i.e. video) and provide a details from a case study. Lastly we compare our approach to the closest method in the literature.

Sift Dimensionality and Multiple Classifiers. Recall that the output of our system is a transformation which can be applied to any input feature vector (i.e., face image) to produce a sifted output intended to uphold a given policy. Our results indicate that the average (across all policies) *PubLoss* monotonically decreases while the average *PrivLoss* monotonically increases as the number of sift dimensions exposed to classifiers grows. This is reasonable since very low dimensional sifts do not carry enough information to classify public attributes while high dimensional sifts provide an increased risk of information leakage.

In our evaluation we adopt a conservative threshold, and set the acceptable *PrivLoss* to inferences that are 10% better than chance (i.e., maximal allowed private classification accuracy is 60%). Given this constrain we find that an output sift dimensionality of $K = 5$, and $\lambda = 1$ yield the best average tradeoffs across policies (with one public and one private attribute accross all tested classifiers). Figure 3 provides examples of our system's output for two policies (which use the same attributes in exchanged public/private order) in which classification accuracy is shown as a function of sift dimensionality.

From an adversarial standpoint, the output of our system represents an 'un-sifting' challenge which can be tackled with any available tool(s). In general we find that for low dimensional sifts, classifier accuracies are similar despite differences in the algorithmic machinery used for inference; however as the sift dimensionality grows the classifiers increasingly differ in performance — when we look across classifiers using the 90 simple policy combinations possible with one public and one private attribute, we find that 5 dimensional sifts have an avg. public attribute accuracy standard deviation of 3.86% and an avg. private attribute accuracy standard deviation of 3.77%; whereas 15 dimensional sifts have significantly larger deviations as avg. public attribute accuracy standard deviation is 8.25% and avg. private attribute accuracy standard deviation is 14.16%. Another interesting observation is that the linear-SVM and kernel-SVM classifiers consistently produced the lowest *PubLoss* while the linear-SVM and randomForest classifiers produced the highest *PrivLoss*. The high performance of linear-SVM is not surprising given the linear nature of our PPLS algorithm.

Policy Results. We evaluated sifts created for each of our 90 policies (10 attributes paired with all others, excluding self matches) using each of our 5 classification methods. For each policy, we report the lowest *PubLoss* and highest *PrivLoss* obtained across all 5 classifiers in Figure 4. In these matrices, the attribute enumeration used in the rows and columns is: (1) Male - *M*, (2) Attractive Female - *AF*, (3) White - *W*, (4) Youth - *Y*, (5) Smiling - *S*, (6) Frowning - *F*, (7) No Eyewear - *nE*, (8) Obstructed Forehead - *OF*, (9) No Beard - *nB*, and (10) Outdoors - *O*. Recall that the *PubLoss* results are relative to the achievable accuracies reported in the third column of Table 2.

Our results indicate that we can create sifts that provide strong privacy and minimize utility losses at ($K = 5$ dimensions) for the majority of policies we tested (average *PubLoss* = 6.49 and *PrivLoss* = 5.17). This is a significant finding which highlights the potential of policy driven privacy and utility balance in sensor contexts!

Performance Impacting Factors Based on our analysis we find that the PPLS algorithm is able to produce high performing sifts as long as there are not significant statistical interactions between the public and private attributes. This is to be expected given the structure of the problem we are trying to solve. In the extreme case, if we consider a policy which includes the same attribute in its public and private set it seems obvious that any privacy enforcing algorithm will have a hard time balancing between utility and privacy since obscuring the private attribute prevents recognition of the [same] public attribute.

To formalize the intuition above we use two quantitative measures to capture the levels of statistical interactions in policies: correlation and overlap. Correlation is the traditional statistical measure of the probabilistic dependence between two random variables (in our case attributes). Overlap is a metric we introduce to describe the degree to which two attributes occupy the same regions in feature space. Overlap is computed as in equation (6) and normalized to 1 across our 90 policies. The Correlation and Overlap matrices in Figure 4 show the correlation and overlap for each attribute pair in our tested policies.

$$find \sum\sum \max_{w} \left[cov(Xw, Y^{+})^2 * cov(Xw, Y^{-})^2 \right] \qquad (6)$$
$$s.t. \quad w^{\top} w = 1$$

To help illustrate correlation and overlap we provide a set of examples from our analysis. Consider the attributes Male and No Beard. These attributes are highly correlated ($r = -.72$). Male and Attractive Female are another highly correlated attribute pair ($r = -.66$). Using our domain knowledge we can reason about these numerical dependencies as follows: if you know about the presence of facial hair (i.e., No Beard is false) then Maleness is easily predicted, similarly if an individual is an ttractive Female it is highly unlikely that they are Male.

Although correlations provide a key insight into the interactions between attributes a deeper level of understanding is obtained by investigating overlap. Returning to our examples, Male and No Beard have an overlap (.29) which is less than half of the overlap of Male and Attractive Female (.72). The reason for this is that No Beard is a relatively localized attribute (i.e., pixels around around the mouth/chin) and does not depend on features in many of the regions used to determine Male-ness. Conversely, Attractive Female and Male have high overlap because they are determined using many of the same feature regions (i.e., eyebrows, nose, bangs) as can be seen in Figure 5.

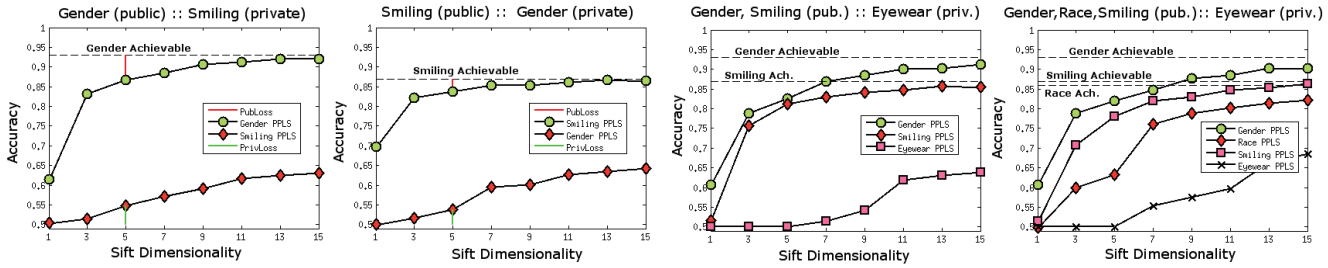


Figure 3: Left: *PubLoss* **and** *PrivLoss* **performance (classification accuracy) as a function of sift dimensionality for two simple policies. Right:** *PubLoss* **and** *PrivLoss* **performance for complex policies. In all figures, the lowest** *PubLoss* **and highest** *PrivLoss* **is reported across all five classifiers. Dashed lines represent the maximum achievable accuracies using raw (unsifted) data which serve as upper bounds for** *PubLoss* **performance.**

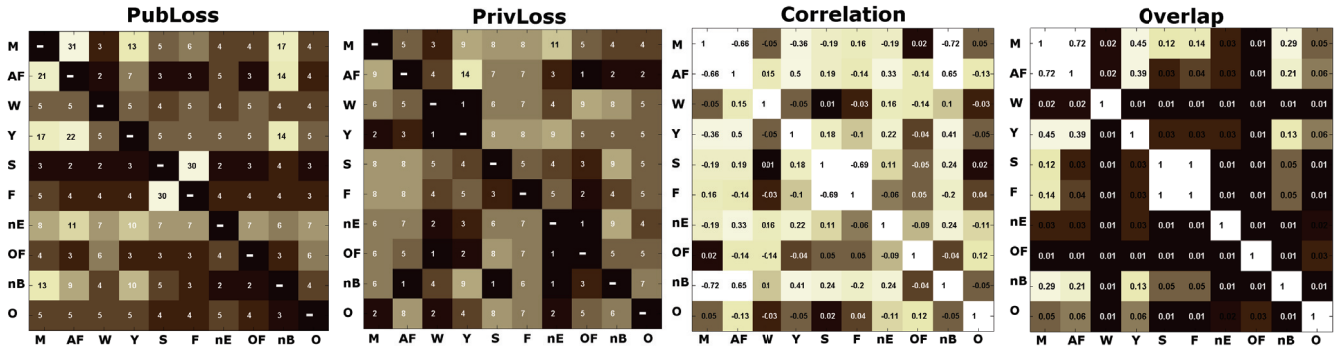


Figure 4: *PubLoss*, *PrivLoss*, **Correlation, and Overlap matrices for our 90 simple policy combinations. Rows denote the public attribute, columns represent the private attribute, while cells represent policies which combine the row and column attributes. In the case of** *PubLoss* **and** *PrivLoss* **lower values are desirable as they indicate minimal utility and privacy sacrifices respectively. Correlation values are shown using absolute values and higher cell values indicate significant information linkages between attributes. Lastly, high Overlap values indicate that attributes occupy the same regions in feature space.**

Intuitively highly correlated attributes with significant overlap should prevent utility and privacy balance. This is indeed what we see when we match up the results of the *PubLoss* and *PrivLoss* matrices with the correlation and overlap matrices (Figure 4).

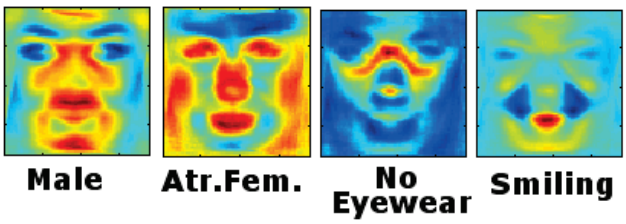


Figure 5: The image features (from the red component of the raw RGB values) which most strongly covary with several attributes (red values indicate strong positive correlations, blue values indicate strong negative correlations).

A regression analysis model attributes 63% of the *PubLoss* estimate to correlation and 37% to overlap. In the case of *PrivLoss* the weights correspond to 67% and 33% for correlation and overlap respectively. Furthermore, using regression we find that correlation alone as a predictor leads to a SSE (sum of squared error) term which is 315% larger than if correlation and overlap are used together. These findings suggest that correlation and overlap should be considered together when analyzing sifting performance.

Extensions: Complex Policies. Although the bulk of our analysis has focused on policies in which there is one public and one private attribute, our algorithm can be augmented to support multiple public and multiple private attributes. To illustrate the potential for more complex policies, we modified the PPLS objective function to produce the largest average gap between multiple public and multiple private attribute covariances relative to data features.

$$\begin{array}{ll} find & \max_w [avg(cov(Xw, Y_1^+), cov(Xw, Y_2^+), ...)^2 \\ . & \quad -\lambda * avg(cov(Xw, Y_1^-), cov(Xw, Y_2^-), ...)^2] \\ s.t. & w^\top w = 1 \end{array}$$

Using this averaging method, we were able to find high performing masks for various policies which include several public and/or several private attributes. An example of two such policies is provided in Figure 3.

Complex policies can include arbitrary ratios of (public:private) 1:2, 1:3, 2:1, 2:2, 3:1 (i.e., public: *'Male'* + *'Smiling'*, private: *'White'* + *'Youth'*). The number of complex policy combinations is very large, however in our tests using (35 complex policies) we found that the same principles from Section 7 apply. Just as in the case of simple policies correlations and overlap have a big impact on *PubLoss* and *PrivLoss*. In general as polices grow to include many attributes the likelihood of significant correlation/overlap grows thus increasing the chance of diminishing utility and privacy balance. A more detail analysis of complex policies is a deep topic which is certainly an attractive target for future work.

Extensions: Streaming Content. So far we have focused our analysis on static sensor samples (i.e., still photos), however dynamic data (i.e. streaming video) is also of importance. To evaluate the SensorSift scheme in a dynamic context we used the Talking Face dataset [3]. The data consists of 5000 frames taken from a 29fps video of a person engaged in a natural conversation lasting roughly 200 seconds. Using the annotations provided from the dataset we first cropped the face region from each frame. Next we

extracted image features as described in Section 5. Subsequently we used the Face.com [16] labeling tool to determine the frames in which the individual was smiling.

As evaluation, we applied the sift for the policy Male (public) Smiling (private) to concatenated sets of 10 sequential frames (identified as smiling) together prior to computing *PubLoss* and *PrivLoss*. As an additional pre-processing step we made sure that the sequences of frames we used as our concatenated samples did not occur at the boundaries of smiling events). We find that the PrivLoss accuracy increases by only 2.3% while PubLoss accuracy decreased by 4.5% (using 5 dimensional sifts and a $\lambda = 5$).

This is an encouraging result and suggests that the SensorSift technique can be applied to dynamic sensor contexts, however, in instances where samples are accumulated over longer time sequences (i.e., days, months) the dynamics of privacy exposure are likely to change and so will the optimal parameter settings for sift output dimensionality and privacy emphasis (λ). This is certainly an important area for further research as dynamic sensing becomes more ubiquitous (i.e. Microsoft Face Tracking Software Development Kit in Kinect for Windows [2]).

Comparison to Related Work. The most similar publication to our present effort is a recent article by Whitehill and Movellan [18] which uses image filters (applied to a face dataset) to intentionally decrease discriminability for one classification task while preserving discriminiability for another (smiling and gender). This work uses a ratio of discriminability metrics (based on Fisher's Linear Discriminant Analysis) to perform a type of linear feature selection. Perhaps the most significant difference between [18] and SensorSift is that the authors evaluate the quality of their privacy filters against human judgments whereas we target automated inferences.

To compare against [18] we used the methods and demo dataset provided on their website. The dataset consists of 870 grayscale images (16x16 pixel 'face patches'). It also provides labels for smiling and gender thus enabling analysis of two policies (1) gender (public) : smiling (private), and (2) smiling (private) : gender (public).

For each policy we evaluated 3 different combinations of training and testing data splits (using different 80% 20% splits of training and testing respectively). For each combination we generated 100 discriminability filters using the provided algorithm (total of 300 filters for each policy) and subsequently used a linear SVM classifier to evaluate their quality. We found that even though these filters were reported to prevent successful human judgement on the private attribtue, even the best filter we found was not able to deter machine inference.

In particular the lowest private attribute accuracy for the gender (public) smiling (private) policy was 81.21% (average 86.32%). Conversely the lowest private attribtue accuracy for the smiling (public) gender (private) policy was 77.65% (average 83.12%). The public attribute accuracy decreased by 4% on average relative to classification performance on unfilted (raw) images.

8. RELATED WORK

Below we touch on the related literature in the broader context of balancing utility and privacy and subsequently describe efforts withing the more focused area of face-privacy on which we base our experimental evaluation.

Utility Privacy Balance. There are several classes of approaches which have been proposed for finding a utility and privacy balance in database and/or information sharing contexts. Among these, the developments in differential privacy and cryptographic techniques are only remotely connected to our present discussion as they focus on statistical databases and very limited homomorphic encryption respectively [7]. More pertinent are the systems based approaches which typically use proxies/brokers for uploading user generated content prior to sharing with third-parties. These approaches use access control lists, privacy rule recommendation, and trace audit functions; while they help frame key design principles, they do not provide quantitative obfuscation algorithms beyond degradation of information resolution (typically for location data) [14].

Lastly, there are several papers which have looked at the question of privacy and utility balance from a trust modeling and information theoretic perspectives [5, 6]. While these are very valuable problem characterizations which we use to motivate our formal analysis, we go beyond their framing and develop an algorithmic defense tool which we apply to a real world problem. Furthermore we introduce an information processing scheme for embedding our algorithm into a trusted platform for potential deployment in smart sensor applications.

Previous Approaches to Face Privacy. Prior work on preserving the privacy of face images and videos has been almost exclusively focused on data transformations aimed at identity anonymization. The methods range from selectively masking or blurring regions of the face or the whole face [4], perturbing the face ROI (region of interest) with noise through lossy encoding [13], and face averaging schemes (k-Same, and its variants [8, 15]) aimed at providing k-anonymity guarantees (each de-identified face relates ambiguously to at least k other faces). Whereas these methods emphasize recognition deterrence their methods of limiting information exposure are unconstrained in what face attribute details they perturb. The only notable exception is the multi-factor (ε, k)-map algorithm [8] which demonstrates a selective ability to enhance the representations of facial expressions in k-anonymity de-identified faces, however this approach does not consider privacy granularity below the level of identity protection.

9. DISCUSSION

Our approach aims to mitigate the emerging privacy threats posed by automated reasoning applied to harvested digital traces of personal activity by using algorithmic defenses that enable selective information exposure – private information should remain private, while other non-private information can be harvested and used. We believe that this is a promising approach towards offering quantitative privacy assurances in the rapidly growing market of smart sensing applications.

A critical strength of the SensorSift design is the built in support for innovation by future application developers. We provide an algorithm for generating sifting transformations which can be used by developers to unlock access to novel data features. As long as the sifting transformation functions can be verified to yield minimal sensitive information exposure our system will allow it to operate over the sensor data. This ability to dynamically generate and verify novel privacy respecting data access functions enables flexibility and provides a way to keep up with the rapidly evolving needs of software providers.

Limitations. We stress that, as with many systems, privacy is not binary. Indeed, it may be impossible to achieve absolute privacy in any useful sensor-based system. Our goal, therefore, is to explore new directions for increasing privacy for sensor-based systems while flexibly supporting the desired functionality.

An important point to consider is that multiple applications may request different privacy views (i.e., sift functions) of the image data. In the present work, we do not consider collusion between applications – two applications may be able to combine their func-

tions to reconstruct information greater than that granted to each application alone. We do note, however, that some simple measures can be used to protect against collusion (e.g., apply SensorSift to all the applications running on a system in unity rather than to each application by itself, or only allow one application access to facial attributes over some period of time).

A second potential weakness of our approach is that adversaries may have additional knowledge sources at their disposal which can reveal private information that SensorSift is unable to counteract. Our goal is to explore how to protect against unauthorized privacy disclosures from the sensed data itself, not to defend against auxiliary information sources. Indeed, auxiliary information can almost always break any privacy or anonymity-preserving system. As an extreme example, suppose the private attribute is race but that the application asks the user to complete a biographical form – which includes race – during the application installation process.

Third, our approach leverages classification metrics to verify that the data exposed to applications does not reveal significant information about private attributes; it is possible that future machine learning tools can significantly outperform our benchmarks. To mitigate this evolving algorithmic threat, our scheme uses an ensemble of multiple machine classification tools which span the space of state of the art linear and non-linear methods. Further, the design is meant to support plug in modules so that new classifiers can be added on demand to enrich the privacy metrics.

10. CONCLUSION

Given the growing demand for interactive systems, the low cost of computational resources, and the proliferation of sophisticated sensors (in public/private locations and mobile devices) digital traces of our identities and activity patterns are becoming increasingly accessible to third parties with analytics capabilities. Thus, although sensor systems enhance the quality and availability of digital information which can aid technical innovation, they also give rise to security risks and cause privacy tensions. To address these concerns we proposed a theoretical framework for quantitative balance between utility and privacy though policy based control of sensor data exposure.

In our analysis we found promising results when we evaluated the PPLS algorithm within the context of optical sensing and automated face understanding. However, the algorithm we introduce is general, as it exploits the statistical properties of the data; and in the future it would be exciting to evaluate SensorSift in other sensor contexts.

Acknowledgements

We would like to thank the members of the UW SecurityLab and Dr. Daniel Halperin for their insightful feedback during the writing process. This work was supported in part by the Intel Science and Technology Center for Pervasive Computing and NSF Grant CNS-0846065.

References

[1] Inside google project glass part 1, 2012. http://www.fastcompany.com/1838801/exclusive-inside-google-x-project-glass-steve-lee.

[2] Kinect for windows sdk, 2012. http://www.microsoft.com/en-us/kinectforwindows/develop/new.aspx.

[3] Talking face video, 2012. http://www-prima.inrialpes.fr/FGnet/data/01-TalkingFace/talking_face.html.

[4] Mukhtaj S. Barhm, Nidal Qwasmi, Faisal Z. Qureshi, and Khalil El-Khatib. Negotiating privacy preferences in video surveillance systems. In *IEA/AIE*, volume 6704 of *Lecture Notes in Computer Science*, pages 511âĂŞ–521, 2011.

[5] Supriyo Chakraborty, Haksoo Choi, and Mani B. Srivastava. Demystifying privacy in sensory data: A qoi based approach. In *2011 IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops)*, pages 38 –43, 2011.

[6] Supriyo Chakraborty, Haksoo Choi Zainul Charbiwala, Kasturi Rangan Raghavan, and Mani B. Srivastava. Balancing behavioral privacy and information utility in sensory data flows. In *Preprint*, 2012.

[7] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st annual ACM symposium on Theory of computing*, STOC '09, pages 169–178, 2009.

[8] R. Gross, L. Sweeney, F. de la Torre, and S. Baker. Semi-supervised learning of multi-factor models for face de-identification. In *CVPR*, pages 1–8, 2008.

[9] David Kotz, Sasikanth Avancha, and Amit Baxi. A privacy framework for mobile health and home-care systems.

[10] N. Kumar, P. N. Belhumeur, and S. K. Nayar. Facetracer: A search engine for large collections of images with faces. In *ECCV*, pages 340–353, 2008.

[11] N. Kumar, A. C. Berg, P. N. Belhumeur, and S. K. Nayar. Attribute and simile classifiers for face verification. In *ICCV*, 2009.

[12] Shan Li and David Sarno. Advertisers start using facial recognition to tailor pitches, 2011. http://www.latimes.com/business/la-fi-facial-recognition-20110821,0,7327487.story.

[13] Isabel Martnez-ponte, Xavier Desurmont, Jerome Meessen, and Jean franois Delaigle. Robust human face hiding ensuring privacy. In *WIAMIS*, 2005.

[14] Min Mun, Shuai Hao, Nilesh Mishra, Katie Shilton, Jeff Burke, Deborah Estrin, Mark Hansen, and Ramesh Govindan. Personal data vaults: a locus of control for personal data streams. In *Proceedings of the 6th International COnference*, Co-NEXT '10, pages 17:1–17:12, 2010.

[15] Elaine M. Newton, Latanya Sweeney, and Bradley Malin. Preserving privacy by de-identifying face images. *IEEE Trans. Knowl. Data Eng*, 17(2):232–243, 2005.

[16] Yaniv Taigman and Lior Wolf. Leveraging billions of faces to overcome performance barriers in unconstrained face recognition, August 2011.

[17] Cajo J. F. ter Braak and Sijmen de Jong. The objective function of partial least squares regression. *Journal of Chemometrics*, 12(1):41–54, 1998.

[18] Jacob Whitehill and Javier Movellan. Discriminately decreasing discriminability with learned image filters. In *CVPR*, 2012.

Biometric Authentication on a Mobile Device: A Study of User Effort, Error and Task Disruption

Shari Trewin[1], Cal Swart[1], Larry Koved[1], Jacquelyn Martino[1], Kapil Singh[1], Shay Ben-David[2]
[1]IBM T.J. Watson Research Center
{trewin, cals, koved, jmartino, kapil}@us.ibm.com
[2]IBM Research Haifa
bendavid@il.ibm.com

ABSTRACT

We examine three biometric authentication modalities – voice, face and gesture – as well as password entry, on a mobile device, to explore the relative demands on user time, effort, error and task disruption. Our laboratory study provided observations of user actions, strategies, and reactions to the authentication methods. Face and voice biometrics conditions were faster than password entry. Speaking a PIN was the fastest for biometric sample entry, but short-term memory recall was better in the face verification condition. None of the authentication conditions were considered very usable. In conditions that combined two biometric entry methods, the time to acquire the biometric samples was shorter than if acquired separately but they were very unpopular and had high memory task error rates. These quantitative results demonstrate cognitive and motor differences between biometric authentication modalities, and inform policy decisions in selecting authentication methods.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Authentication; H.5.2 [**User Interfaces**]: Interaction styles

General Terms

Security, Human Factors

Keywords

Authentication, mobile, biometric, usability

1. INTRODUCTION

Mobile devices are rapidly becoming a key computing platform, transforming how people access business and personal information. Access to business data from mobile devices requires secure authentication, but traditional password schemes based on a mix of alphanumerics and symbols are cumbersome and unpopular, leading users to avoid accessing business data on their personal devices altogether [7].

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

The rich set of input sensors on mobile devices, including cameras, microphones, touch screens, and GPS, enable sophisticated multi-media interactions. Biometric authentication methods using these sensors could offer a natural alternative to password schemes, since the sensors are familiar and already used for a variety of mobile tasks.

User frustration with password-based authentication on mobile devices demonstrates that a high level of usability must be achieved for a mobile authentication technique to be accepted. As biometric recognition algorithms continue to improve, the user experience will be an increasingly critical factor in the success of such techniques.

In this paper, we explore authentication techniques on mobile devices from the users' point of view. We study three biometric authentication modalities - voice, face and gesture, and combinations of voice with face and gesture. A typical 8-character password condition is included as a baseline.

This study is the first to measure user action times for authentication using different biometrics on a mobile device. It provides insight into user performance when using these techniques under favorable conditions.

The study examined:

1. The time taken to provide an authentication sample (password, biometric, or two biometrics);
2. Error rates in providing a sample of suitable quality for analysis by verification algorithms;
3. The impact of the user actions required for authentication on performance in a memory recall task; and
4. User reactions to the authentication methods.

To allow for comparison between authentication methods, the voice and gesture conditions use the same 8-digit authentication token. We find that speaking was the fastest biometric authentication method, but taking a photograph supported better performance in the memory recall task. Speaker verification was considered less usable than password, face and gesture (writing an 8-digit PIN). Combination conditions – simultaneously entering two biometric samples – were very unpopular. Failure rates were not significantly different among single conditions, but combining methods led to high error rates.

2. BACKGROUND AND RELATED WORK

2.1 The Mobile Context and Authentication

The proliferation of smartphones, such as those based on Apple, Android, Microsoft and Blackberry technologies, is rapidly changing the nature of interactive computing. Much

of this is driven by the multitude of digital sensors embedded within these devices, including GPS, touch screens, cameras and microphones. As a result, peoples' expectations around ease of use of mobile devices are changing.

Simple gestures (e.g. Android screen lock pattern), graphical passwords [11], and biometric authentication [22] are beginning to emerge as alternative mobile authentication mechanisms, but passwords and PINs remain the most common methods used today. Corporate use of mobile devices is frequently dictating the use of password strength policies, derived from desktop password policies, for device screen unlock. A typical company password policy requires a mix of alphabetic and numeric or symbol characters [7].

Bao et al [7] measured the time to type an 8-character, mixed-case alphanumeric password on desktops and mobile phones. On mobile devices with soft keyboards, entry of compliant passwords often requires the user to switch between different keyboard layouts. They found that while participants typed the password at 17wpm on a desktop computer, they only achieved a mean of 6wpm on their own phones. Mobile device users are acutely aware of this additional effort. Their participants found password typing on a mobile phone so onerous that they avoided business data access on their phones because it would have required a corporate-compliant device unlock password.

Even in desktop environments, users often select poor quality passwords [12][13]. The perceived effort of entering passwords on mobile devices will encourage further password simplification, for example placing non-alphabetic characters only at the beginning or end of the password. Recall aids such as writing down passwords and physically attaching them to devices [31] pose additional security risks for password authentication in a mobile context.

Interaction with mobile devices tends to be brief and interruption driven [24][25]. As a result, mobile devices have been caching the security credentials in the device to make it easier for users to authenticate. The result is that mobile devices have effectively become authentication tokens (e.g., [1][17]). Given that mobile devices are often borrowed [18], and perceived to be more frequently lost or stolen [23], users' personal and business resources are at greater risk of being lost or compromised.

2.2 Mobile Biometric Authentication

Biometric authentication is a well-studied area of research. Physical biometrics, such as face, voice and signature, are the most commonly used forms. Biometrics authentication systems have been evaluated against a rich set of metrics that incorporate both performance and usability aspects [10]. User attitudes have been explored [14][19][30], but relatively little attention has been paid to empirical comparison of the usability of biometric authentication methods. Toledano et. al's usability evaluation of multimodal (non-mobile) biometric authentication systems [32] is a notable exception. It proposes a testing framework for biometric usability analysis that uses ISO usability factors (i.e., effectiveness, efficiency and satisfaction) for evaluation.

We believe that the era of using biometric authentication for mobile devices is imminent. People are now accustomed to talking into small mobile devices, and seeing themselves through the device camera. As the quality of sensors and processing power of mobile devices improves, mobile biometric authentication has become a realistic proposition. Diverse usage environments, including poor lighting, motion/vibration, and ambient noise, pose significant challenges to biometric recognition algorithms. Research has explored algorithms suitable for use on mobile devices [16][21], and for processing face and voice data gathered in noisy mobile environments [2], or with low resolution cameras [29]. Researchers have also investigated fusion of multiple biometrics to compensate for loss of quality in one modality [3][8][34]. For example, Hazen et. al [15] explored the combination of face and voice recognition on an iPAQ device, finding significant improvements in recognition accuracy compared to either biometric alone. Krawczyk and Jain [20] explored signature and voice modalities on a tablet device. All of these studies focused on recognition performance. Combining biometrics also supports 'liveness testing' – the ability to differentiate a live user from a spoof. Efforts in this space [28] have focused both on biometric analysis and custom user challenges.

We are not aware of any existing comparison of user experience in password and biometric authentication on mobile devices, prior to this study. Little is known about the usability of these methods in comparison to each other, and to passwords. Further, little is known about the ease with which users can simultaneously provide two biometric samples, to support efficient multi-factor authentication.

2.3 Working Memory

When accessing information on mobile devices, authentication is an interruption in the user's primary task flow, and a disruption to working memory. The greater the demands on working memory from the authentication process, the greater the risk of forgetting aspects of the task at hand.

Tasks performed on mobile devices, and in particular those performed in the context of a business activity, involve multi-step procedures. In light of the brief nature of the tasks performed on these mobile devices [25], in this study we raise the question of how much of an impact authentication challenges have on users' working memory and thus on reliable task completion. Prior studies indicate that there is an impact, particularly just before task completion (e.g. [33]). Part of the present study is to assess the recall impact due to authentication modality, or combination of modalities, on a memory recall task in the absence of recall cues (e.g., [4]).

Working memory is the mental process by which information is temporarily stored and manipulated in the performance of complex cognitive tasks. The capacity of working memory is limited, and varies between individuals. Models of working memory describing a multi-component system including a phonological loop and visuo-spatial scratch pad were introduced in the early 1970s [5] and have decades of empirical support. The 'phonological loop' stores and rehearses verbal and other auditory information, while the 'visuo-spatial scratch pad' manipulates visual images. Information stored in working memory fades, or 'decays' over time. Subvocal (or even vocal) articulation is a commonly used memory strategy, in which an individual repeatedly subvocally verbalizes and hears an item in order to rehearse it and maintain its activation in working memory. Verbal authentication methods could interfere with this process.

3. USABILITY STUDY

Three different forms of user action for biometric authentication, password entry, and two combinations were exam-

ined in six experimental conditions described below. All voice and gesture conditions used the same authentication phrase, '35793579', providing a memorable consistent value across both modalities, and an audio sample long enough to be acceptable for an automated speaker verification technology. A repeated 4-digit sequence was used to increase memorability while still using a variety of gestures and speech sounds. Password entry was included as a reference point.

This paper uses the terms 'user action' and 'taking action' to refer to the actions taken by the user in providing an authentication sample (biometric or password). As authentication algorithms improve, these user actions will be an important determinant of technology acceptance. This study assumes a zero false rejection rate (FRR), the ideal scenario for a legitimate user.

The six experimental conditions were as follows:

1. Password: Enter an alphanumeric password using the built-in on-screen keyboard. In the spirit of typical corporate password policies, the easy to remember 8-character password *securit3* was used.
2. Voice: The user must speak the password phrase "three five seven nine three five seven nine".
3. Face: The user must take a photograph of their face using the front-facing camera.
4. Gesture: The user must write '35793579' on the screen with their finger.
5. Face+Voice: The user must say "three five seven nine three five seven nine" while simultaneously lining up their face and taking a photograph.
6. Gesture+Voice: The user must say "three five seven nine three five seven nine" while simultaneously writing the digits '35793579' on the screen with their finger.

3.1 Participants

Participants were 30 employees (13 women) of a large technology corporation, unconnected to the project, having 1.5 to 45 years with the company. They were recruited through email lists and personal contacts, and were given a small compensation. Twenty-nine have experience using a smartphone. Six use multiple smartphones. Twenty-one have used a tablet device with the iPad being the most common device and one month to two years of experience. Five used a smartphone and three used a tablet device to access protected company information, where policy required a mobile device screen lock password of at least 8 characters, including both alphabetic and numeric or symbol characters.

All participants had experience with password and PIN as an authentication method. Five occasionally used on-screen signature, four regularly used other types of gesture id and one occasionally did. Six occasionally used face id (3) or voice id (3). Ten occasionally used fingerprint while one regularly did. Some participants' work had at some time involved taking or analyzing facial images for verification (4), recording or analyzing speech samples for voice or speaker verification (7), or collecting or analyzing gestures (3).

3.2 Apparatus and Materials

3.2.1 Hardware

Participants used a Motorola Xoom touch screen tablet with 1GHz Dual Core processor, 1GB RAM, 32GB memory, and 10.1in HD widescreen 1280x800 resolution display. The tablet was running Android version 3.2.1 (Honeycomb).

Figure 1: Face Authentication Screen

It measured 249.1mm x 167.8mm x 12.9mm (HxWxD) and weighed 708g. We used the built-in 2MP front-facing camera with automatic focus, located in the top center of the long side of the tablet, making landscape the natural device orientation for taking a photograph. The microphone was centered on the lower long edge.

3.2.2 Client Software

An Android app was developed in HTML, CSS and JavaScript, using PhoneGap v1.0.0rc2 [2] with custom-built audio, camera and gesture capture extensions. The app recorded photographs, gestures, audio recordings, and a time-stamped log of user and system actions.

Each condition presented a different authentication screen. Figure 1 shows the Face authentication screen. The gesture screen presented a plain white writing area with the instruction "write PIN". The Voice authentication screen showed a glowing microphone with the text "Say the PIN", and a counter showing the recording time.

In each condition, three practice trials were given. In Face and Face+Voice conditions, the software also instructed users to lower the device between attempts, so as to practice the full process of positioning the device.

After the practice trials, the software presented a series of memory task trials. This simulates the situation where a user performing a task must authenticate before they can complete the task. The memory task presented a randomly generated three-digit number and a two-character measurement unit randomly selected from 10 options, for example 'The value is 512mg'. Tapping an 'Authenticate' button activated the authentication screen for the current condition. After taking action, participants were asked "What is the value?", and entered their response using the on-screen keyboard. Buttons for 'Done' and 'Forgotten' were available. No feedback on response accuracy was provided.

In all conditions, users could start to take action as soon as the authentication screen was displayed. Specifically, the on-screen keyboard was automatically displayed, voice recording was on, the camera was active, or the gesture capture was active, as appropriate. Users pressed a button to complete their authentication action. Placement of these buttons was influenced by the expected user action. For example, the button on the face authentication screen was placed in the lower right, for convenient thumb activation while holding up the tablet with two hands (see Figure 1).

Each sample resulting from a user action (password or biometric) was immediately checked by the server. This simulates a likely usage scenario where an organization policy is

to control access to its information rather than authenticating the local device.

If the sample quality was not acceptable (as defined below), an error message was displayed, and the user was returned to the authentication screen. After three failed attempts, the software moved on to the next trial.

3.2.3 Acceptance Criteria

No automated verification was performed. Instead, a server on the local network assessed password, voice, face and gesture input quality. Voice input samples were quality checked by a remote server. Acceptance of the sample depended on passing the following simple quality checks:

1. Password: The password (*securit3*) was typed correctly. The error message provided for incorrect passwords was "Authentication failed, please try again"
2. Voice: The user provided a sample containing at least 1.5s of speech content with a speech level > 1000 (32767 indicates full dynamic range) and a signal-to-noise ratio $>= 20$dB. The error message provided for failed voice samples was "voice sample too short, too noisy, or no voice found, please try again"
3. Face: The photograph was accepted when it contained a face, as determined by the VeriLook SDK. This ensured that pictures of the ceiling, fuzzy images, and partially hidden faces would not be accepted. The error message provided for failed face samples was "no face found, please try again"
4. Gesture: A gesture is comprised of one or more strokes, each made up of line segments connecting recorded finger positions on the screen. The gesture was accepted when it contained at least 20 line segments. The error message provided for failed gesture samples was "gesture too short, please try again"
5. Face + Voice: The image and voice sample both met the quality criteria as above.
6. Gesture + Voice: The gesture and voice sample both met the quality criteria as above.

This approach establishes a best case scenario for the user, in which their biometric is always recognized so long as they provide a usable sample (FRR=0). The laboratory environment, tightly-specified task and presence of a researcher combined to ensure that participants performed the authentication correctly, minimizing false acceptances. Samples were manually examined for conformance.

3.2.4 Other Materials

The 10-question System Usability Scale (SUS) assessment tool [9] was used to gather subjective impressions of the usability of each authentication action. The word 'system' in the standard questionnaire was replaced with the word 'method'. After pilot testing, questions 5 and 6 were appended with further explanation shown in italics below:

5. I found the various functions in this method were well integrated (*I could remember the values in the task easily after authenticating*)
6. I thought there was too much inconsistency in this method (*I got different results for the same authentication input*)

Responses to each question are given on a five-point scale ranging from 'Strongly disagree' to 'Strongly agree'. An overall SUS score is a value between 0 and 100, where a higher value indicates a more usable method. An average SUS Score is 68 [27]. Sauro [27] analyzed over 500 studies using the SUS, allowing a raw SUS score to be transformed into a percentile, while Bangor et al [6] proposed an A-F grading scale, allowing for easy interpretation. Raw scores, percentiles and grades are all reported here.

An 11th question, using the same response scale, was added: "This method was tiring to use."

Participants were also asked "What did you like or dislike about this method?" A 10-question demographic questionnaire elicited background information including experience authenticating on mobile devices.

3.2.5 Location

Study sessions were conducted in three different interior rooms with overhead fluorescent office lighting; one small office, one larger office, and one 10-person conference room.

3.3 Procedure

After providing informed consent, participants used six different forms of authentication action, presented in random order, and then filled in the demographic questionnaire.

We chose to use a standing position. This makes interaction more challenging because the user must hold the device while operating it, and enabled participants to explore different lighting positions easily. All were advised that they could lean on a desk or a wall, move freely around the room as they wished, and rest at any time.

For each condition, a researcher showed a printed image of the authentication screen and described the user action to be taken. On-screen instructions were also provided. The instructions for taking a photograph were "Authenticate by taking a well-lit photo of your face. Put your nose in the box and use a neutral expression. Press 'done' when you are ready to take the photo." When Face was combined with Voice, participants were instructed to "Authenticate by saying the PIN AND taking a well-lit photo of your face. You can speak while lining up your face, or speak first and then take the photo. Put your nose in the box and use a neutral expression. Press 'done' when you are finished speaking AND are ready to take the photo." In the Gesture+Voice condition, the instructions were: "Authenticate by saying the PIN AND writing it on the screen with your finger. You can write and speak at the same time, or in any order you choose. Press 'done' when you have finished both writing and speaking".

Participants executed 3 practice trials then went on to a set of 8 memory task trials. They were not told that the system was not performing automated verification of their face/voice or gesture. A researcher observed participant actions, comments, position, and method of holding the tablet device. In voice conditions, participants were corrected by the researcher if they did not say the correct phrase. It was not possible to see their gestures during the sessions.

After completing each condition, participants sat down to fill in the usability questionnaire. This provided an opportunity to rest. The instruction given for the usability evaluation questionnaire was:

"Where these questions ask about "the method" we mean the authentication method you just used, within the context of the scenarios where you are trying to remember a number and unit. This includes the experience of sometimes having to repeat your actions to get a good sample, or correct

Table 1: Biometric performance summary

Condition	Failure to Enroll (FTE) % of participants	Failure to Acquire (FTA) % of attempts	User action time per error-free attempt (median sec)
Password	0.0	4.2	7.46
Voice	3.4	0.5	5.15
Face	6.9	3.1	5.55
Gesture	0.0	0.0	8.10
Face+Voice	10.3	21.3	7.63
Gesture+Voice	3.4	13.6	9.91

an error. For example, 'learning to use the method' means learning how to use it accurately, to avoid the need to repeat."

3.4 Data Available

Two participants ran out of time and attempted only 5 of the 6 conditions. A further 16 trials are missing due to technical problems. Three participants did not complete all conditions because they were unable to provide either face or voice samples that passed the acceptance test (see below for further details). Finally, one participant abandoned the Gesture+Voice condition after 2 scenarios due to frustration with that method.

Data from one participant, whose comments indicated that he was testing the authentication mechanisms rather than performing the requested tasks, were discarded.

Authentication attempts were coded as follows:

1. Success: The participant performed authentication correctly and was successful. (1229 samples)
2. Minor error: The participant performed well enough to succeed but may have included additional speech or corrected errors. (43 samples)
3. Error: The user attempted to provide the correct authentication but failed, for example a password with errors, a fuzzy picture, or a speech sample that did not meet the quality check. (100 samples)
4. Noncompliance: The user did not perform authentication correctly, for example speaking the value to be memorized ('529mg') instead of the PIN, saying nothing, or writing a squiggle. (35 samples)
5. Technical error: The sample was unusable due to technical problems. (14 samples, all empty or clipped speech files)

Technical errors and noncompliant attempts were excluded from the analysis.

4. RESULTS

4.1 Failure to Enroll (FTE)

The 'Failure to Enroll' metric (FTE) used in biometric usability research [10] is intended to identify the proportion of individuals who would never be able to use a biometric system. Table 1 summarizes the failure to enroll (FTE) rates for each condition.

Two of the 29 participants found that the Face condition did not work for them – they were not able to take a picture in which the face verification engine could locate their face. These participants contributed no data for the Face or Face+Voice conditions. One of these participants always wears dark, light blocking glasses.

One participant was not successful with the Voice condition – their voice samples did not meet the threshold for signal-to-noise ratio. They contributed no data for the Voice, Face+Voice and Gesture+Voice conditions.

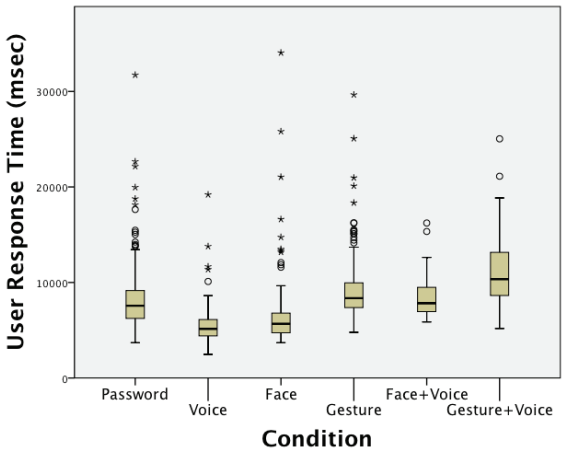


Figure 2: User response time by authentication condition

4.2 Failure to Acquire (FTA)

The 'Failure to Acquire' (FTA) metric [10] is used in biometric usability research to measure failure to provide a sample of sufficient quality. In this study it captures failures where a participant provides a sample that does not meet the predefined quality criteria. For biometric samples, such samples do not contain good enough data on which verification algorithms can operate.

1372 user actions were analyzed, of which 92.7% were successful. Table 1 summarizes the percentage of these attempts that were unsuccessful, in each condition. Face+Voice had the highest FTA rate, at 21.3%. A one-way ANOVA indicated a significant effect of condition on success ($F(5,1366) = 27.249$, $p<0.001$), with post-hoc pairwise comparisons using Bonferroni corrections indicating that FTA values for Face+Voice and Gesture+Voice are significantly different from each other ($p=0.013$) and all other conditions ($p<= 0.001$). The differences between the remaining conditions are not statistically significant.

One participant abandoned the Gesture+Voice condition after 2 scenarios, in which he succeeded only once out of 6 attempts, despite having success in the practice. If he had completed all 8 scenarios with the same low success rate, the overall FTA rate for Gesture+Voice would have been 18.7%.

4.3 User Action Time

User action time is time spent by the user taking action to provide the sample for authentication. It does not include processing time spent verifying the sample quality, performing authentication, or server response delays.

This measure was calculated for the 1229 successful trials (coded as 'Success'), with 184-221 samples per condition. Figure 2 illustrates the distribution of user response times in each condition. Voice authentication was both fast and consistent, with few outlier values. As shown in Table 1, the voice sample was fastest with a median of 5.15 seconds

Table 2: Memory task performance summary

Condition	Memory task preparation time (median sec)	Memory task (% success)
Password	4.3	73
Voice	5.4	76
Face	3.9	85
Gesture	4.2	72
Face+Voice	5.3	71
Gesture+Voice	5.7	65

and taking a photo took 5.55 seconds. The other conditions all took 7.46 seconds or more, with Gesture+Voice being the slowest at 9.9 seconds. The data are not normally distributed, so the Friedman test was used as a non-parametric alternative to a one-way ANOVA with repeated measures. There was a statistically significant difference in user action time depending on the authentication method ($\chi^2(5) = 430.339$, P<0.001). Post-hoc analysis with Wilcoxon Signed Rank tests was conducted. Applying Bonferroni correction, the significance level was set at P<0.003. All pairwise comparisons were statistically significant (P<0.001) with the exception of Password and Face+Voice (Z=-1.128, P=0.259).

4.4 Memory Task

The memory task required participants to enter a three-digit value and two-digit measurement unit they had been shown prior to the authentication action, using the on-screen keyboard. Trials containing technical errors or noncompliant attempts are excluded (N=21), leaving 1277 trials for analysis.

Table 2 shows the median memory task preparation time, defined as the time participants spent viewing the screen that showed the value before proceeding to the authentication screen. This gives an indication of time spent actively memorizing the value. Face had the least time with a median of 3.9s. Using the Friedman test as a non-parametric alternative to a one-way ANOVA with repeated measures, there was a statistically significant difference in preparation time depending on the authentication method ($\chi^2(5) = 81.334$, P<0.001). Post-hoc analysis with Wilcoxon Signed Rank tests was conducted with Bonferroni correction applied, resulting in a significance level set at P<0.003. There was a statistically significant difference between Face and all other conditions except Gesture (Password: Z=-3.121, P=0.002, Voice: Z=-4.297, P<0.001, Gesture: Z=-1.602, P=0.109, Face+Voice: Z=-3.340, P=0.001, Gesture+Voice: Z=-7.447, P<0.001). There was also a statistically significant difference in preparation time between Voice and Gesture (Z=-4.064, P<0.001), with participants spending approximately one second longer in the Voice condition. All other pairwise comparisons were not statistically significant.

In the 1277 memory task trials, the participants entered the correct response 74% of the time. The success rate for the 1204 trials where user action was successful at the first attempt was 75%, while the success rate for the remaining 64 trials was 56%. These memory task failures include typing errors as well as cases where the user pressed the 'Forgot' button, or omitted all or part of the response. Table 2 shows the percentage of correctly completed memory tasks for each condition (Memory task % success). There was an overall statistically significant difference in success

Table 3: System Usability Scale summary

Condition	SUS score	SUS response percentile (approx.)	SUS grade	Fatigue
Password	78%	80^{th}	C	2.5
Voice	66%	40^{th}	D	3.0
Face	75%	76^{th}	C	2.2
Gesture	77%	78^{th}	C	2.4
Face+Voice	46%	8^{th}	F	3.7
Gesture+Voice	50%	13^{th}	F	3.8

depending on the authentication method ($\chi^2(5) = 28.261$, P<0.001). The combined Face+Voice condition was associated with significantly poorer performance than Face or Voice alone (Wilcoxon Signed-Ranks test with Bonferroni correction, significance level P<0.003, Voice: Z=-3.094, P=0.002, Face: Z=-5.000, P<0.001), and the combined Gesture+Voice condition was poorer than Face (Z=-3.299, P=0.001). Other pairwise comparisons were not statistically significant.

4.5 Usability Responses

Table 3 summarizes the overall score, percentile and grade for the System Usability Scale (SUS) for each condition, and level of agreement with the question "This method was tiring to use". These interpretations illustrate that none of the user actions were well liked in the context of the memory task, with grades ranging from C to F. Password, Face and Gesture were rated above the average SUS response value, while the combination conditions lagged behind, with ratings in the 10th percentile of typical responses. The combination conditions were also considered the most tiring to use, while Password, Face and Gesture were not tiring.

In Table 3, ratings from the three participants who experienced failure to enroll (FTE) are included. Excluding all ratings from these participants increases the scores for Face, Voice and Face+Voice by 1-2 points and does not impact the other scores, leading to the same overall assessment.

Participant responses also take into account the processing time used to communicate the sample to the server, assess the quality, and provide a response. Variable, and sometimes long, network delays were observed, and likely influenced these usability results. Median server response times were: Password=0.06s; Voice=2.04s; Face=1.49s; Gesture=0.13s; Face+Voice=4.28s; and Gesture+Voice=3.82s.

4.6 Participant Comments

Participants provided comments both while using the tablet, and in written form after each condition in response to the question "What did you like or dislike about this method?" Conditions were ordered randomly, so participants' first impressions of a biometric may have been in a single or combination condition.

4.6.1 Password

Participants liked the familiarity of password entry, commenting that there was "no need to learn new tricks", it was "comfortable, easy and familiar", "seemed to be the fastest method and easiest to remember the measurements" and "familiar = easy = like!"

However, they did not like that "the input requires many

steps (including switching back and forth between alphabet and number input)". One person commented that "1. Having to switch keyboards affected my memory terribly, 2. As well as having to have a number in it" (the password). Another observed "Keyboards that do not display letters AND numbers simultaneously can be irritating in this scenario." One person found that "Standing and keying in letters/digits is a bit of a challenge, balancing the pad on one hand."

4.6.2 Voice

Only three participants made positive comments, that speaker verification using a spoken number was "natural", "faster than other modes that required an additional biometric", or "easier to use than typing".

Most comments were negative. Nine participants commented that they experienced "Interference between the content of the authentication method and what I needed to remember" or it was "impossibly difficult to remember things after speaking".

Participants also expressed concern about the security aspects of this approach. Five participants commented that speaking a phrase out loud "doesn't feel secure". Participants felt that voice would not be a practical method in real contexts, saying "In real life there would be noise, and interference leading to huge frustration". One participant commented on the volume level required for speech "I learned from the last speech based system to speak more loudly. That helped. I still didn't like it."

The Voice recording user interface also received some criticism, that the timer indicator was "distracting and led to some confusion over how fast I should say the passphrase" and it was "confusing with recording on and off message - not sure if I tapped properly to start voice authentication".

4.6.3 Face

Eleven participants made positive comments that "it was easier to remember the numbers", or "I was able to mentally 'repeat' the value, even as I was taking a picture."

Four found it "easy" or "simple" to take the picture, but nine others complained that positioning the camera was "somewhat annoying", "a bit hard because of the reflection of myself I was getting" or "cumbersome to position the face". Participants commented on the lack of feedback when their face was positioned properly: "I didn't know when it worked well", or "not sure how accurately I need to place my nose in the box on the screen."

Participants took action to get better pictures: "I had to find a solid background and then it worked", or "I found a better lit spot in the room". Several participants felt uncomfortable taking a picture of themselves: "I have to suspend the fact that I might not like the picture", "felt too much like I was taking a vanity photo."

4.6.4 Gesture

Some participants found the gesture condition "fun", "fast", "easy to use", "fairly automatic", and "an intuitive way of entering passwords". One participant observed that "I could easily see what the system was getting from me (vs. audio where I don't hear the recording)".

However, in the context of the memory task, it was "mechanically easy to use but cognitively difficult", and "still easy to forget the value". Eight participants commented that it was difficult to remember the memory task value while writing the phrase, but four considered it easier than other conditions, for example "the writing of numbers is like a pattern which makes remembering the other number easier", and "I could use muttering to remember the codes". One suggested a shorter password, while another observed that it would have been easier if the phrase was a word.

4.6.5 Face+Voice

Only two positive comments were made about the Face+ Voice condition, that it had "simplicity" and provided a "double degree of security".

Seven participants commented on difficulty with the memory recall task, for example "I had to invent memory aids to remember the number and units to key after authenticating."

Eleven participants commented on the physical difficulty of the required actions. For example it was "cumbersome", "requires too much coordination", was "very annoying trying to get the camera at the right angle to get a photo", and "felt like a lot of work". Other comments included "Positioning nose in square on screen is not easy; once nose is in position scanning the screen for 'done' button resulted in moving my face", "I disliked having to center my nose in the target area – I seem to move the tablet about quite a bit without thinking about it and had to make an effort", "My arms get tired holding the tablet up and aligning it for a face shot", and "tilting the screen (both horizontally and vertically) seemed counter-intuitive – my first inclination to tilt it up or left was consistently wrong (moving my nose further away). Over time I overcame this with practice."

A further five felt that the method was not working correctly. Saying the voice performance was "erratic" or "didn't work well", or "too slow", and "Had a few failures when I moved around possibly because of lighting".

As with the Face condition, participants also mentioned a dislike of looking at their own images: "didn't like seeing myself at such close-up!" and "it makes me self-conscious".

4.6.6 Gesture+Voice

No positive comments were made about the combined Gesture+Voice condition. Eight participants commented on difficulty with the memory recall task. Seven participants commented that the performance "seemed slow", "the numbers I wrote appeared distorted", and it "did not seem to track the movement of my finger with good resolution".

Participants chose to speak as they wrote, but three commented on the awkwardness of slowing down their natural speech rate to match their writing speed: "Unlike the first experience w/ writing (alone) this seemed too slow – I guess because the voice channel is so much faster than the gesture feedback", "I can speak much faster than I can write so having to do both was off putting (because I was very aware of the 'slowness') whereas when I was just writing it 'felt' just right."

Some participants considered this condition "horrible", with "WAY too much distraction".

4.7 Researcher Observations

As participants performed the study, they often moved around the room. Some participants paced as they worked, while most stood or leaned against a wall or desk. Those who paced, stopped pacing to take a photograph, but continued pacing while entering a password, writing or speaking.

The tablet was normally held at chest or belly height. Participants were observed to switch positions as they became tired.

The method of holding the tablet was also strongly influenced by the experimental condition. When using the camera, 23 participants held it with two hands, one at each side, and held it up in front of their face, lowering it again afterwards. When tapping in a password, participants often held the device with one hand spread underneath, whereas the most common position for gesture was to hold the device with the left hand at the left side. When speaking, participants did not move the tablet, and 22 held it in their left hand.

While practicing with the camera, participants moved around the room and experimented with different tablet angles and positions, then used a single location and position throughout the remainder of the study. Taking a face picture was made more difficult by the distraction of seeing their reflection in the shiny screen, under the strong overhead lighting.

Even when the voice was clearly audible to a person in the room, the signal-to-noise ratio was sometimes low. Some participants needed to speak more loudly than was comfortable in order to reduce error rates. Those who experienced problems with the voice condition reacted first by speaking more loudly or slowly. Only two looked for or asked about the microphone location, and two moved the tablet closer to their mouth.

When voice was combined with face or gesture, participants appeared to speak with lower volume and have a tendency for their voice to trail off. This reduced the signal-to-noise ratio, causing voice quality failures.

The participants were highly motivated to perform well on the memory task, and employed techniques to help them remember the value and unit, including speaking the value aloud, or thinking of a mnemonic to help them remember. These techniques were used more often in conditions involving speech.

5. DISCUSSION

These data provide an understanding of the relative user effort required by the different authentication mechanisms under quiet, well-lit, stable conditions and may be representative of environments such as an office or home location. Work is ongoing on robust authentication algorithms that are effective in a broad range of environments that are noisy, low lighting, or involve movement (e.g., walking, public and private transportation), etc. and multi-factor biometric authentication. Privacy considerations may be addressed by cancellable biometrics [26].

The interfaces for biometric and password acquisition used here were simple. With the exception of a screen orientation to facilitate self portrait photos (landscape), we did not attempt to compensate for any perceived shortcomings of the device (e.g., reflections on the display surface, alternative keyboard layouts to minimize changing between alphabetic and numeric/symbol layouts). Our participants were novice users, and performance improvements with practice could be expected. Further field studies in natural environments with more experienced users are needed to provide a more complete understanding, including learning effects.

5.1 Time to provide an authentication sample

Clearly the Face and Voice conditions were faster than the Password and Gesture conditions. The Gesture entry was significantly slower than any of the other conditions, although that may be related to the substantial software lag time in responding to drawing on the touch screen. On average, the Face and Voice conditions had a 2.0-2.5 sec. lower user action time than the 7.5 sec. in the password condition. Participants were able to provide dual biometrics in less time than sequential entry of the same two biometrics, but with higher acquisition error rates. The error-free Face+Voice condition time was comparable to error-free password typing. Where there is a failure to provide an acceptable biometric sample, the overall time would quickly rise, underscoring the importance of an authentication interface that minimizes user error through appropriate feedback to the user, and recognition algorithms that can operate on real-world samples with minimal error. For the Face conditions, once participants found a place with good lighting, they tended to stay in that position. In outdoor or highly populated environments such as public transport, additional actions, and time, would be required to find a suitable location, and biometrics will sometimes not be appropriate.

5.2 Ability to provide a quality sample

With minimal instruction and very little practice, 90% of participants were able to use all of the biometric methods well enough to provide a sample that met the quality criteria. However, there were three participants who could not use one of the biometric modalities. In two cases, the reasons for these failures are not clear, and will be explored in further work. This failure rate underscores the importance of having multiple modalities for authenticating, with a reliable fallback method to support critical access scenarios.

The dual conditions had error rates much higher than the sum of the individual error rates. High error rates negate the benefit of dual conditions by increasing the overall time to acquire beyond the time that would be required for single biometrics in sequence. There are multiple possible explanations for the higher error rates. Given the low error rate in the Gesture condition, but high lag time for displaying the gesture, the high error rates for Gesture+Voice may be due to fading off in the voice sample. Poor performance on the Voice+Face condition may be due to the cognitive demand of a task involving two disparate modalities. Practice may reduce these dual condition error rates, but this remains to be empirically tested.

In future work, we will examine the quality and consistency of biometric samples provided by the participants, and the performance of verification algorithms on this data set.

5.3 Impact on the memory recall task

In contrast to prior work that examined password typing time on a mobile device [7], this study presented authentication within a task that demanded short term memory recall. Authentication 'failure' due to a poor quality sample, led to a steep drop in task success, from 74% to 47%, confirming the challenge of the task and the disruptive nature of authentication. Perhaps because of this cost of failure, participants actively employed memory recall strategies to boost their task performance.

Face authentication, the only condition that involved no password or PIN, supported the highest memory task perfor-

mance. Using the same authentication prompt in all other conditions, no significant difference was found between voice and gesture modalities. Combination modalities produced significantly poorer performance.

Participants spent significantly longer on the trial screen that presented the memory task in the Voice condition, compared to Gesture or Face. This may be indicative of additional effort invested in memorization of the values when in conditions that involve speech. These results underscore the importance of carefully choosing authentication points that least interfere with user task flow.

Further work should examine the impact of using different kinds of spoken/gestural material such as spoken phrases, or abstract gestures, and user-selected vs. system-selected items. This would separate users' reactions to the method of authentication from the content of the authentication prompt. Although system generated prompts may increase the cognitive load on the user.

One possibility would be to allow users to combine prompted speech with any other speech of their choosing. Participants could, for example, have chosen to say something like "526mg 35793579 526mg", ensuring liveness while allowing them to verbalize any information in working memory. This may actually help with their task, rather than hinder it. In contexts where the task is known, prompts should be designed so as not to interfere with the task content.

5.4 User reactions

User responses to the SUS were low, with grades ranging from C to F. As one participant put it "Authentication is never fun". Interestingly, the Voice condition was faster, less error prone, did not suffer very long server delays, and supported relatively high task success, yet received only a 'D' grade for usability from participants. Although participants perceived it as interfering with their ability to perform the memory task, this was not reflected in their results. Authentication prompts that are very different in nature to the task context may reduce such interference to some extent, and should be explored in future studies.

From observations of users during the study, many were not comfortable with the speech volume required for sample acceptance. Sample quality and naturalness of speech need to be carefully balanced.

User reactions to Face authentication were mixed, with some commenting that the process of taking a photograph was cumbersome, while others found it easy. Further work into appropriate user feedback to make it easier to take a good quality photograph with a tablet device in varied locations is needed.

Dual biometric conditions were considered fatiguing and less usable by participants. However, these conditions also involved variable, and sometimes long, server delays. Server response time should be more tightly controlled in future work, to allow for separation of the impact of user action times, modalities and prompts.

6. CONCLUSIONS

We report a laboratory study of the usability of three biometric authentication modalities on a tablet device within the context of a memory task, independent of the performance of biometric verification algorithms. Speaker, face and gesture verification, as well as password entry, were compared using 8-digit written and spoken PIN codes, under six single and dual-biometric conditions. The study identifies usability issues and biometric performance requirements that can serve as a focus for research.

Each biometric modality has unique strengths and weaknesses, and has the potential to improve on the Password approach. Face and Voice are fast but not universally usable. Gesture is reliably performed and worked for everyone, but a much shorter gesture would be needed to achieve a competitive time, posing a challenge to gesture recognition algorithms. The memory task context provides further insight into the broader impact of authentication, and demonstrates a significant advantage for Face, and a lesser advantage for Voice in supporting memory task performance.

However, the Voice condition was considered less usable than Password, Face and Gesture. Speaking at a comfortable level did not always meet the voice sample quality threshold, indicating a requirement to operate with a lower threshold. Participants also reported interference with the memory task that was not reflected in their performance. They maintained high performance by using sophisticated memorization strategies, as indicated by their comments and differences in authentication preparation time.

Using face recognition also posed challenges for participants, even in good conditions. Careful user interface design is needed to overcome issues with screen reflection and provide feedback for proper alignment.

The conditions that combined two biometric authentication modalities were disliked by the participants, had higher Failure To Acquire and lower performance on the memory recall task. This suggests that combined sample collection for biometric fusion is not necessarily preferable to collecting individual samples.

Providing a face or voice biometric to a mobile device seems to be a natural extension of normal device usage requiring no special setup or extra hardware. Software developments such as built-in face recognition are opening further opportunities to streamline the user experience of mobile authentication. This study demonstrates a complex set of trade-offs in selecting and using biometric authentication methods on mobile devices, even in quiet, well-lit conditions. Studies like this one can help to identify critical research challenges for biometric verification algorithms, in addition to design challenges for mobile authentication user interfaces. The goal is to improve on the notoriously cumbersome password method, leading to mobile biometric authentication that is both secure and usable.

7. ACKNOWLEDGEMENTS

We thank the study participants, and Bonnie E. John, Rachel L. K. Bellamy, John C. Thomas, Nalini Ratha, David Nahamoo, Ron Hoory, Hagai Aronowitz, and Amir Geva for valuable feedback, and technical contributions.

8. REFERENCES

[1] A. Adams and M. A. Sasse. Users are not the enemy: Why users compromise computer security mechanisms and how to take remedial measures. *Communications of the ACM*, 42(12):40–46, Dec. 1999.

[2] Adobe Systems Inc. PhoneGap. `http://phonegap.com`.

[3] G. Aggarwal, N. K. Ratha, R. M. Bolle, and R. Chellappa. Multi-biometric cohort analysis for

biometric fusion. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Las Vegas, NV, 2008.

[4] E. Altmann and G. Trafton. Task interruption: Disruptive effects and the role of cues. In *Proceedings of the 26th Annual Conference of the Cognitive Science Society*, Chicago, IL, 2004.

[5] A. Baddeley and G. Hitch. Working memory. In G. Bower, editor, *Recent Advances in Learning and Motivation*. Academic Press, 1974.

[6] A. Bangor, P. T. Kortum, and J. T. Miller. An empirical evaluation of the system usability scale. *International Journal of Human-Computer Interaction*, 2008.

[7] P. Bao, J. Pierce, S. Whittaker, and S. Zhai. Smart phone use by non-mobile business users. In *MobileHCI*, Stockholm, Sweden, 2011.

[8] J. Basak, K. Kate, V. Tyagi, and N. Ratha. QPLC : A novel multimodal biometric score fusion method. *CVPR Workshop on Biometrics*, 2010.

[9] J. Brooke. *SUS: A quick and dirty usability scale*, pages 189–194. Taylor and Francis, 1996.

[10] L. Coventry. Usable biometrics. In L. F. Cranor and S. Garfinkel, editors, *Security and Usability: Designing Secure Systems that People can Use*. O'Reilly Books, 2005.

[11] P. Dunphy, A. P. Heiner, and N. Asokan. A closer look at recognition-based graphical passwords on mobile devices. In *SOUPS*, Redmond, WA, 2010.

[12] D. Florencio and C. Herley. A large-scale study of web password habits. In *WWW*, Banff, Canada, 2007.

[13] D. Florêncio and C. Herley. Where do security policies come from? In *SOUPS*, Redmond, WA, 2010.

[14] N. Gunson, D. Marshall, F. McInnes, and M. Jack. Usability evaluation of voiceprint authentication in automated telephone banking: Sentences versus digits. *Interacting with Computers*, 23(1):57–69, Jan. 2011.

[15] T. J. Hazen, E. Weinstein, B. Heisele, A. Park, and J. Ming. Multimodal face and speaker identification for mobile devices. In R. I. Hammoud, B. R. Abidi, and M. A. Abidi, editors, *Face Biometrics for Personal Identification: Multi-Sensory Multi-Modal Systems*. Springer, 2007.

[16] Y. Ijiri, M. Sakuragi, and S. Lao. Security management for mobile devices by face recognition. In *Proceedings of the 7th International Conference on Mobile Data Management (MDM)*, Nara, Japan, 2006.

[17] N. Jackson. Infographic: How Mobile Phones Are Replacing Our Credit Cards, 2011. `http://www.theatlantic.com/technology/archive/2011/07/infographic-how-mobile-phones-are-replacing-our-credit-cards/241703/`.

[18] M. Jakobsson, E. Shi, P. Golle, and R. Chow. Implicit authentication for mobile devices. In *HotSec*, Montreal, Canada, 2009.

[19] L. A. Jones, A. I. Antón, and J. B. Earp. Towards understanding user perceptions of authentication technologies. In *Proceedings of the ACM Workshop on Privacy in Electronic Society*, Alexandria, VA, 2007.

[20] S. Krawczyk and A. K. Jain. Securing electronic medical records using biometric authentication. In *Proceedings of the 5th International Conference on Audio- and Video-Based Biometric Person Authentication (AVBPA)*, Hilton Rye Town, NY, 2005.

[21] S. Kurkovsky, T. Carpenter, and C. MacDonald. Experiments with simple iris recognition for mobile phones. In *Proceedings of the 2010 Seventh International Conference on Information Technology: New Generations (ITNG)*, Las Vegas, NV, 2010.

[22] M. Lee. Google Turns to Face Detection With Samsung to Take On Apple Speech Parser, 2011. `http://www.bloomberg.com/news/2011-10-19/google-turns-to-face-detection-to-take-on-apple-iphone-s-speech-technology.html`.

[23] M. Lennon. One in Three Experience Mobile Device Loss or Theft. Do People in 'Party Cities' Lose More Phones?, 2011. `http://www.securityweek.com/one-three-experience-mobile-device-loss-or-theft-do-people-party-cities-lose-more-phones`.

[24] S. F. Nagata. Multitasking and interruptions during mobile web tasks. In *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, Denver, CO, 2003.

[25] A. Oulasvirta, S. Tamminen, V. Roto, and J. Kuorelahti. Interaction in 4-second bursts: the fragmented nature of attentional resources in mobile hci. In *CHI*, Portland, OR, 2005.

[26] N. K. Ratha, S. Chikkerur, J. H. Connell, and R. M. Bolle. Generating cancelable fingerprint templates. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(4):561–572, Apr. 2007.

[27] J. Sauro. Measuring usability with the System Usability Scale (SUS), 2011. `http://www.measuringusability.com/sus.php`.

[28] S. A. Schuckers, R. Derakhshani, S. Parthasardhi, and L. A. Hornak. Liveness detection in biometric devices. In *Electrical Engineering Handbook, 3rd edition*. CRC Press, 2006.

[29] Q. Tao and R. N. J. Veldhuis. Biometric authentication for a mobile personal device. In *Proceedings of the 3rd Annual International Conference on Mobile and Ubiquitous Systems: Networking & Services*, San Jose, CA, July 2006.

[30] R. Tassabehji and M. A. Kamala. Improving e-banking security with biometrics: modelling user attitudes and acceptance. In *Proceedings of the 3rd International Conference on New Technologies, Mobility and Security (NTMS)*, Cairo, Egypt, 2009.

[31] B. Tognazzini. Design for usability. In L. F. Cranor and S. Garfinkel, editors, *Security and Usability: Designing Secure Systems that People can Use*. O'Reilly Books, 2005.

[32] D. T. Toledano, R. Fernández Pozo, A. Hernández Trapote, and L. Hernández Gómez. Usability evaluation of multi-modal biometric verification systems. *Interacting with Computers*, 18(5):1101–1122, Sept. 2006.

[33] J. G. Trafton and C. M. Monk. Task interruptions. In D. A. Boehm-Davis, editor, *Reviews of Human Factors and Ergonomics*. 2008.

[34] V. Tyagi and N. Ratha. Biometrics score fusion through discriminative training. *CVPR Workshop on Biometrics*, 2011.

BetterAuth: Web Authentication Revisited*

Martin Johns
SAP Research
martin.johns@sap.com

Sebastian Lekies
SAP Research
sebastian.lekies@sap.com

Bastian Braun
University of Passau
bb@sec.uni-passau.de

Benjamin Flesch
SAP Research
benjamin.flesch@sap.com

ABSTRACT

This paper presents "BetterAuth", an authentication protocol for Web applications. Its design is based on the experiences of two decades with the Web. BetterAuth addresses existing attacks on Web authentication, ranging from network attacks to Cross-site Request Forgery up to Phishing. Furthermore, the protocol can be realized completely in standard JavaScript. This allows Web applications an early adoption, even in a situation with limited browser support.

1. INTRODUCTION

1.1 Motivation

The current state of password-based authentication on the Web is a mess. If used in its default configuration without additional protection measures, today's Web authentication almost appears to be an exercise in demonstrating how an authentication process should *not* be realized, showcasing severe flaws, such as, sending the password in cleartext over the wire, allowing untrusted parties to create arbitrary authenticated requests, or exposing the authentication credentials to potentially malicious code. While there have been first stabs in the direction of improving Web-based password authentication, previous approaches expose at least one of the following problems:

Web authentication differs from most other authentication scenarios: It exposes many characteristics that resemble properties from security protocols. However, it lacks a security protocol's rigorous enforcement of message sequence and integrity, resulting, for instance, in enabling the insertion of messages in authenticated workflows via Cross-site Request Forgery. Hence, proposals that approach Web authentication purely from a protocol perspective are in danger of solving only a subset of the problems and missing issues that result from the versatile and fragile nature of Web interaction.

*This work was in parts supported by the EU Projects STREWS (FP7-318097) and WebSand (FP7-256964).

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

Furthermore, the vast majority of proposed improvements require fundamental changes both in the browser as well as in the client/server interaction. Hence, without Web browser support Web applications cannot benefit from the potential security benefits. This leads to a chicken/egg problem, as there is no early adopter path for motivated developers, which in turn could encourage the browser vendors to natively implement the mechanism.

In consequence, the basic process of password authentication on the Web has not significantly changed since the day in which the `type="password"` attribute was introduced to HTML.

1.2 Contribution & Organisation

In this paper, we propose BetterAuth, a password-based authentication scheme that is tailored to fit the Web's security requirements and mitigate the flaws of the current scheme. Our approach has the following properties:

- Unlike related approaches [42, 40, 1, 2, 8, 37], BetterAuth spans the full authentication lifecycle, consisting of both the initial authentication process and the ongoing authentication tracking. This allows both a lightweight, consistent design as well as robust, end-to-end security guarantees.

- Furthermore, BetterAuth is secure by default. The developer does not need to enable security properties explicitly. Instead, all security goals are met due to inherent properties of the scheme. In consequence, in its default state, BetterAuth transparently addresses many weaknesses of the established approach, including password sniffing, session credential theft, session fixation, and cross-site request forgery.

- Finally, even while being suited to be adopted as a native capability of Web browsers, BetterAuth can be implemented completely in standard JavaScript. This enables sites to use the scheme today without having to wait for the browser vendors to catch up. This potentially enables a viable, transitional phase, in which only a subset of deployed Web browsers support the scheme natively.

Organisation: The remainder of the paper is structured as follows: First, we summarize the current state of Web-based password authentication, both from the attacker as well as the developer's point of view (Sec. 2). Then, we describe BetterAuth, our improved authentication scheme

(Sec. 3) and report on our experiences in practically implementing the protocol (Sec. 4). An evaluation on security, performance, and limitations is given in Sec. 5. Before we conclude in Sec. 7, we discus related work (Sec. 6).

2. THE CURRENT STATE OF WEB-BASED PASSWORD AUTHENTICATION

The basic process of authenticating against Web applications has not changed significantly since the early days of the Web. In the following sections, we show how the current state of Web authentication came to be. First, we discuss the bare-bones authentication mechanism that is in use by the vast majority of all existing Web applications (see Sec. 2.1). Please note, that in this description, we omit all potential security measures. We simply show how Web authentication would look like, if implemented as is and how little security is provided by default. Then, in Sections 2.2 and 2.3, we revisit attacks on Web authentication and the countermeasures which were introduced to mitigate these threats.

Also, please note, that for the remainder of this paper, we restrict the discussion to password-based authentication, and in this respect, even further to the well established practice of form-based authentication (see Sec. 2.1), as virtually all professional Web applications utilize this method.

2.1 The Basics of Web Authentication and Authentication Tracking

The Web authentication process consists of two steps: First, the initial authentication, in which the user provides his user ID and password to the application's server-side. Then, the authenticated state of the user is maintained over the series of following HTTP request/response pairs. The next two sections will explore these two processes.

Initial authentication: In form-based authentication, the user's ID and password are communicated using HTML forms. After the user has entered his credentials, he submits the form. This causes the Web browser to create an HTTP request, which carries the values in the form of GET or POST parameters. In particular, this implies that the password is sent in clear-text to the server. The server compares the submitted user ID and password with its internal records. If the password and ID match with one of its records, the authentication process succeeds and the user's session is promoted to an authenticated state.

Authentication tracking: HTTP is a stateless protocol. Therefore, there is no protocol-level mechanism to promote a usage session into an authenticated state, as there is no inherent session concept. In consequence, application-layer measures for session and authentication tracking had to be introduced. The dominant method to maintain an authenticated state over a series of HTTP requests is to use HTTP cookies for this purpose. An HTTP cookie is a value that is set by a Web server for the Web server's domain. The value is stored by the browser. From this point on, all further requests that are sent to the server's domain carry the cookie value automatically, via the `Cookie`-header. To implement authentication tracking, the Web server sends a cookie to the browser, which signifies the authenticated state of this client. All further requests which are received by the server carrying this cookie value are regarded as being authenticated under the user's identity. Hence, the cookie value is de facto the user's authentication credential. Again, as with the password, this credential is communicated in cleartext. NB: Instead of setting a new cookie, the server could also promote an already existing session identifier (SID) cookie into an authenticated state, thus, making this SID the user's credential.

2.2 Fixing Web Authentication: A History of Band-Aid Solutions and Additive Design

In this section, we briefly revisit documented classes of Web attacks that target either the initial authentication or the authentication tracking process. In addition, we discuss the protective measures that have to be taken by the application developer to mitigate the respective threat.

2.2.1 Network-Based Attacks

As already mentioned in Sec. 2.1, both the user's password as well as the authenticator cookie are communicated in cleartext to the server. This opens the communication to various network-level attacks:

For one, every party that is able to observe the network traffic between the browser and the server can simply sniff the password or cookie value and abuse these credentials under the identity of the user. Furthermore, parties with direct access to the network link can also launch man-in-the-middle attacks, which allows the dynamic modification of HTTP requests and responses.

To counter these threats, the SSL/TLS protocol was introduced, which provides end-to-end confidentiality and integrity guarantees on top of TCP, making the sniffing of authentication credentials infeasible. Furthermore, SSL/TLS provides a PKI-based scheme to prove the server's identity to the user. This way, attempted man-in-the-middle attacks can be mitigated (as long as the user does not choose to ignore the warning dialogues).

SSL Stripping: Most Web applications serve content both encrypted, via HTTPS, as well as unencrypted, via HTTP. Unfortunately, if the user does not explicitly specify the protocol when he accesses a Web page, browsers default to HTTP. In consequence, in the majority of all cases, the first HTTP request to a server is sent via plain HTTP. This opens a loophole for a network-based man-in-the-middle attacker – the so-called SSL Stripping attacks [26].For this first request, an end-to-end SSL/TLS connection has not been established yet. Thus, the attacker can set himself in between the browser and the server and modify the server's responses. This way, even if the server requires HTTPS for certain operations and tries to redirect the browser accordingly, the attacker can simply remove these redirection attempts from the server's responses, before they reach the client. The client is forced to indefinitely communicate unencrypted.

To combat this problem, the `HSTS` HTTP response header [17] was created. This header tells the browser that from now on for a defined time period, all communication with the server shall be conducted using HTTPS. Under the assumption that the first connection to the server has been done using an attacker-free network path, from that point on the browser will reliably and exclusively use HTTPS to communicate with this server. This way, SSL stripping attempts are made impossible.

Further issues with SSL/TLS: The recent past has shown, that the current state of SSL/TLS is not fully bullet

	Transport	HTTP	Cookie	App
SSL/TLS	X			
HSTS		X		
HTTPonly			X	
Anti CSRF		(X^a)		X
Session Fix.	(X^b)		(X^c)	X
Anti Framing		X^d		X^e

a: Origin header, *b*: Origin-Bound Certs (exp.),
c: Origin flag (exp.), *d*: X-Frame options, *e*: JS-framebuster

Table 1: Overview of countermeasures and their respective implementation levels

proof. For one, the security of HTTPS-based communication heavily relies on the security policies and practice of the Certification Authorities (CAs), that issue the root certificates which are included in Web browsers by default. However, issues in that domain have been reported repeatedly, e.g., unlimited RA certificates have been issued [16] and the internal systems of several CA's have been compromised [10, 9]. As the CA system and its security is out of reach of the application's developers and operators, the current approach offers severely limited options to mitigate such threats.

2.2.2 Issues Related to Cookie-Based Authentication Tracking

As discussed above, after the initial authentication process, the cookie value becomes the user's authentication credential. However, HTTP cookies have not been designed with security in mind and were never intended to be used for this purpose.

Session hijacking through cookie theft: For one, based on the fact that the existence of the cookie value in a request suffices that the request is recognized to be authenticated, every party that can obtain this value is able to send arbitrary authenticated requests under the identity of the user. As, by default, the cookie value is sent in cleartext, every party with access to the network can sniff the value for future abuse. While SSL/TLS protects against this threat, many sites only protect the login page with SSL/TLS and then revert back to plain HTTP [19], leaving the cookie exposed.

Even in the existence of an uncompromised SSL/TLS connection, the cookie is readable by default through JavaScript via the `document.cookie` property. Hence, a simple Cross-site Scripting (XSS) vulnerability allows to leak the cookie's value to the adversary. To counter this threat, browser vendors introduced the `HTTPonly`-flag [28], which hides the cookie value from JavaScript. This flag has to be set explicitly by the developer to mark the authentication cookie.

Session Fixation: The `HTTPonly`-flag only prevents read access to the cookie value. However, an attacker is still able to set or overwrite cookie values. Hence, if he is able to set cookies for an attacked domain in the user's browser, he can launch a session fixation attack in which he tricks the application to reuse a value controlled by the attacker as the user's authentication token. Possible scenarios, in which attackers are able to set cookies for foreign domains include XSS, HTTP header injection [23], or insecure subdomains [22].

While this problem is partially addressed with currently experimental browser features [6, 2], the only reliable way for an application to mitigate this attack, is to renew the cookie's value each time the authorization level of the user changes [21].

Cross-site Request Forgery By default, the browser attaches all cookie values that belong to a given origin to every outgoing HTTP request to the corresponding site. However, due to the hypertext background of the Web, several HTTP-tags, such as `img`, `script`, or `iframe`, have the inherent ability to create cross-domain HTTP requests. Regardless of the actual origin of these elements, the browser attaches the target domain's cookies to all HTTP requests that are created this way. This circumstance leads to an attack vector known as Cross-site Request Forgery (CSRF): It is possible for any Web site which is rendered in the user's browser to send authenticated HTTP requests to all other Web sites, which currently maintain an authentication context with the browser.

To prevent third parties abusing this capability to initiate state-changing actions under the user's identity, the developer has to protect all sensitive interfaces of his application. This can be done either using secret nonces [33] or through strict checking of the `origin` request header [3].

Clickjacking: While being only partially related to authentication tracking, Clickjacking [15] (also known as "UI Redressing") is a class of attacks in the CSRF family. Clickjacking exploits the fact, that due to the cookie rules, foreign sites can load authenticated, crossdomain content into `iframes`. Using cascading style sheets, these iframes can be hidden from the user (e.g, by making them completely transparent) and, thus, the user can be tricked to interact with them via clicks or drag'n'drop.

To protect users from such attacks, the developer has to utilize JavaScript framebusting code [35] or the `X-Frame-Options` response header [27].

2.2.3 Phishing

A further serious threat is known by the term "Phishing" [30]. Phishing attacks aim to steal the user's password through simple decoy: A site under the control of the attacker imitates domain name and design of the target Web site. The user is tricked to enter his password into the forged site under the assumption that he interacts with the legitimate application. As HTTP transports the password in cleartext to the communication partner, the attacker is able to obtain and abuse it. A similar approach is followed by "Pharming" [39], a variant of phishing, which utilizes compromised DNS responses.

Due to its social engineering component, there is no straight forward technical solution to combat phishing, as long as the passwords are still sent over the wire. To mitigate the threat, browsers currently check visited URLs for known phishing sites and warn if such a page is accessed [14].

2.3 Summary

To sum the up the previous sections: The current praxis of Web application authentication and authentication tracking is secure if (and only if) the following holds:

- The password is transmitted over an uncompromised SSL/TLS connection in which the authenticity of the Web server has been verified. This requires among others robust defense against SSL-stripping attacks [26], e.g., utilizing the `HSTS` HTTP response header [17].

- All further requests, both belonging to the current session as well as all future sessions, are transmitted over an uncompromised SSL/TLS connection, as long as the authenticated cookie is valid.
- The authenticated cookie is secured against JavaScript read access by the HTTPonly cookie attribute [28].
- The value of the authenticated cookie is changed every time the authorization level of the user changes to combat potential session fixation vulnerabilities [21].
- State-changing interfaces are secured against CSRF using server-side checking of security nonces [33] or strict enforcement of matching `origin` HTTP response header [3].
- UI redressing attacks are avoided by framing prevention [27, 35].

All these measures have to be explicitly introduced and are realized at different positions and abstraction levels within the application architecture, spanning from securing the low-level transport layer via SSL to application layer anti-CSRF prevention (see Table 1 for an overview). Furthermore, even mitigation measures that are positioned at the same level within the application architecture often have to be implemented at separate places in the application's code.

And even if all these factors hold, the basic interaction pattern is still susceptible to phishing attacks, as the current scheme requires sending the password to the server as part of each login process.

3. PROTOCOL DESIGN

As discussed above, the current state exposes numerous security shortcomings. In this section, we present BetterAuth, an improved password scheme which is tailored to the Web's inherent characteristics and addresses the identified problems of the current scheme.

3.1 Design goals

Before we explain the technical aspects of BetterAuth, we briefly state our design goals:

Secure by default: BetterAuth is designed to mitigate the weaknesses of the current approach (see Sec. 2). In particular, these security goals are realized without explicit enabling steps by the developer.

No mandatory reliance on non-existing browser features: BetterAuth is designed in a fashion that allows an implementation for today's browsers. This allows an immediate deployment without the need to wait for browser vendors to implement native support.

No security regression: Regardless of the form of implementation (browser-based or pure JavaScript), BetterAuth has to be at least as secure as the current approach. This means a (re-)introduction of security problems, which are not currently present, is not acceptable.

3.2 High-level overview

Our proposed scheme consists of two steps, implemented as subprotocols:

An *initial mutual authentication protocol* with integrated key negotiation: The browser and the server both prove their knowledge of the password and jointly generate a per-session, shared secret which is used for further authentication tracking.

And an *authentication tracking scheme* which is based on request signing: Every further request from the browser to the server is signed using the freshly generated shared secret, if the request satisfies certain criteria (see Sec. 3.5 for details). Only requests with such a signature are regarded by the server as authenticated.

In the following sections, we give details on the realization of the two subprotocols.

3.3 Initial mutual authentication

As motivated above, one of BetterAuth's pillars is a mutual authentication step, which results in a shared cryptographic session key. Such mutual authentication schemes have received considerable attention in the past. In the given scenario both parties already share a textual secret (i.e., the password). Hence, a suiting choice for this task is a member of the password authenticated key exchange (PAKE) family [12]. PAKE protocols utilize well established cryptographic building blocks, such as the Diffie-Hellman key creation, and protect the communication against active network attackers using the pre-shared password.

While various protocols match our requirements, we selected [32] for our implementation, a scheme which is currently under active standardization by the IETF and, thus, has the potential for future adoption by the browser vendors. The protocol works as follows (see Figs. 1 and 2):

1. *Initial Handshake:* The browser sends a request targeted at the restricted resource. Along with this request, it sends the user's ID (*UID*, e.g., the user name). This causes the server to create the server-side partial key (*SPK*) for the Diffie-Hellmann key generation. The value is encrypted with the password[1], which has been set for the given UID.
2. *Key exchange:* The encrypted *SPK* is sent back to the browser as part of a 401 response. The browser creates the client-side Diffie-Hellmann partial key (*BPK*). The browser is now able to calculate the session key *SSK* using *SPK* and *BPK*. In addition *BPK* is encrypted with the password and added to the next request to the server.
3. *Mutual authentication:* The browser signs (see Sec. 3.4) the *BPK* carrying request using *SSK*. The server receives the request, calculates *SSK* himself and verifies the request signature. As the browser can only correctly compute *SSK*, if it knows the password, the correctness of the signature is used as authentication proof by the server. Hence, the server sends the restricted resource to the browser. Furthermore, the server also signs the response using *SSK*, to let the browser verify the server's knowledge of the password.

3.4 Request Signing

After the first protocol step has concluded successfully, both parties share a fresh symmetric key *SSK*, which from now on will serve as the basis for authentication tracking. Our authentication tracking mechanism is realized by HMACs [24], a well established Message Authentication Code scheme which utilizes cryptographic hash functions.

The client attaches an HMAC-based signature to all further requests to the server which satisfy the criteria given in Sec. 3.5, closely mimicking the current practice of automatically adding cookie headers to outgoing requests. For GET requests, the URL in a normalized form and selected request

[1]NB: This step can also be done with salted passwords.

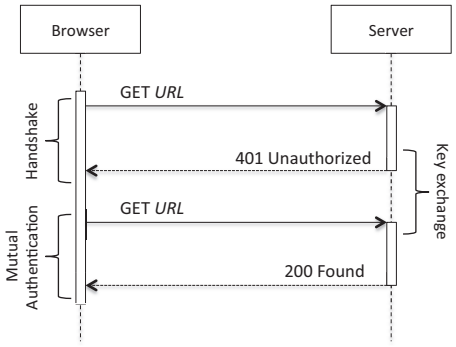


Figure 1: Initial auth. (HTTP communication)

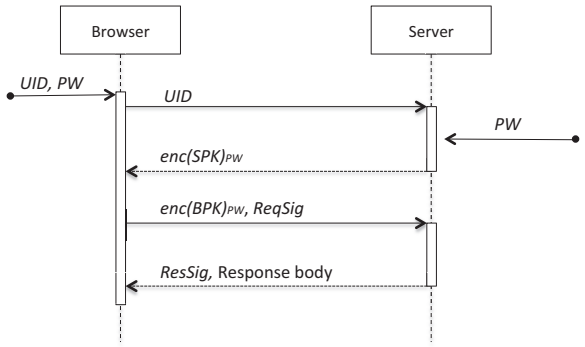


Figure 2: Initial auth. (cryptographic values)

headers are signed, for POST requests, also the POST parameters are included in the signature. Only requests, for which the server can successfully validate the correctness of the HMAC are recognized to be properly authenticated. This way, both the authenticity as well as the integrity of the received requests are ensured.

3.5 Context-Dependent Authentication

As discussed in Sec. 2.2.2, several security problems - most notably Cross-Site Request Forgery - are caused by the fact that currently all requests that originate from an authenticated browser are automatically equipped with the authentication credentials, i.e., the authentication cookies.

Our approach breaks from this troublesome behavior and instead only signs outgoing requests if the request's origin, i.e. the Web page which initiated the request, is already in an authenticated state with the server. Hence, we enforce *in-application authentication tracking*. All requests that are generated in the browser from outside of the Web application, i.e., from third party Web sites, are not signed and, in consequence, not treated as authenticated by the server.

3.6 Public Interfaces

While a strict enforcement of context-dependent authentication would provide robust security guarantees, it is too inflexible to cater to all existing usage patterns of the Web. For example, social Web bookmarking services, such as `delicious.com` provide one-click interfaces to add bookmarks from external pages. Such requests need to be processed in the user's authentication context, as they commit state changing actions to the user's data. However, as they are generated from outside of the Web application's authentication context, they would not receive a signature. Therefore, to enable such scenarios, our approach supports the declaration of *public interfaces*. Such a public interface is a URL for which the server opts in to receive authenticated requests, even if they originate from outside of the application's authentication context. A Web application's public interfaces, if they exist, are communicated to the browser during the initial key exchange using a simple policy format.

3.7 Resulting Authentication Tracking Logic

In consequence, the decision process which requests to sign works as follows:

1. *Test:* Check that the target URL of the request points to a domain, for which currently a valid BetterAuth authentication context exists. Such a context exists, if in the key storage a valid SSK_{app} key could be found, which is assigned to the domain value and that has not yet expired.
2. *Test:* Verify that the request is entitled to be signed. This means, check:
   - Was the request generated *within* the application? This means that the HTML element which was responsible for creating the request (e.g. hyperlink-navigation, form submission, or JavaScript action) is rendered within the browser in the origin of the authenticated application.
   - Or, is the target of the request contained in the applications's list of public interfaces?
3. *Action:* Normalize the request data (Method, URL, selected HTTP headers, request body) and create an HMAC signature using SSK_{app} as signature key.
4. *Action:* Attach the resulting request signature in an `Authorization` header to the request.

4. IMPLEMENTATION

In this section, we present our experiences on practically implementing BetterAuth. We created two different client-side implementations: For one, we built a Firefox browser extension in order to be able to assess how applications would behave, if the BetterAuth-protocol was implemented as a native part of the Web browser (see Sec. 4.1). Furthermore, we implemented BetterAuthcompletely in standard JavaScript (see Sec. 4.2). Using this implementation, Web applications could utilize the protocol during a transitional phase, in which only a subset of browsers support the approach natively.

4.1 Native Implementation

As mentioned above, we approximated a native browser implementation by realizing our approach in the form of a Firefox extension. The extension hooks itself as an observer into the browser's rendering process and monitors the outgoing HTTP requests. Whenever an authentication with a BetterAuth-enabled site is initiated or a request is sent to a domain for which an established BetterAuth authentication context exist, the extension becomes active.

4.1.1 Initial Authentication

If an HTML form is processed during rendering, which is marked with the custom attribute `data-purpose= "better-`

auth" the extension becomes active and the submission process of this form is intercepted: Before submitting the form, the username and password data is retrieved from the request data and used to initiate the BetterAuth-authentication handshake. After receiving the 401 response, the extension removes the password value from the request's data and submits the form.

4.1.2 Authentication Tracking

As discussed in Sec. 3.4, the authentication tracking mechanism mimics the behavior of Web browsers in respect to automatically adding cookie headers to requests that are targeted to the cookie's domain. The extension keeps track of currently active authentication contexts. Whenever a request is targeted towards a domain, for which such an authentication context exists, the extension verifies that the request originated from within this authenticated context or whether the target URL is listed in the application's set of public interfaces (see Sec. 3.5). If one of these conditions is satisfied, the extension transparently signs the outgoing request.

4.2 JavaScript Implementation

Our solution is designed in a fashion that allows to create a pure JavaScript fallback for browsers which do not support our authentication scheme natively. This way, a transitional phase can be supported, which allows developers to already use the mechanism without requiring to provide a separate authentication scheme for legacy browsers. In this section, we document the design of the JavaScript implementation of BetterAuth.

4.2.1 General Approach

The core of the transitional implementation is the replacement of native navigation operations, such as form submissions and page transitions, with a JavaScript initiated loading mechanism. This way, the initial authentication handshake can be executed and all further outgoing requests can be signed by JavaScript before they are sent to the server.

This approach is realized using four distinct elements: A dedicated form handling for the initial authentication (see Sec. 4.2.2), a request signing component (see Sec. 4.2.4), and a dedicated page loader object for pure page transitions (see Sec. 4.2.5). Furthermore, we utilize domain isolation to keep the key material out of reach of potentially untrusted JavaScript code (see Sec. 4.2.3).

4.2.2 Initial Authentication

Implementing the actual initial authentication handshake is straightforward: The BetterAuth-enabled HTML form executes a JavaScript function on form submission which conducts the key exchange handshake. For this purpose, the username and password values are read from the DOM elements. Using the XMLHttpRequest object, the script creates the OPTIONS request to the server's authentication interface. After receiving the server's encrypted Diffie-Hellman key and the optional password salt in the 401 response, JavaScript calculates the browser's Diffie-Hellman key and encrypts it with the password. In addition, after sending the key to the server, the script calculates the session signing key using the two key fragments.

4.2.3 Isolating the Secure Key Storage

As in Sec. 3.1 stated: It is unacceptable for any aspect of our technique to introduce security flaws which are not present in the current state. For this reason, we have to take measures to separate the key material from potentially untrusted JavaScript code.

An implementation of the authentication tracking process requires that the session signing key is handled by standard JavaScript functions. In consequence, a careless implementation would lead to a situation in which an XSS-attack could be used to steal this key and leak it to the adversary. Such an attack would be comparable to XSS-based cookie stealing, which can effectively be mitigated using the HTTPonly cookie flag. Hence, to avoid the introduction of security regression, we have to ensure that the key material is kept out of reach of untrusted parties.

To achieve this, we leverage the guarantees provided by the same-origin policy [34] and the postMessage API [38]: First, we introduce a separate subdomain, which is responsible to handle and store the signing key. This domain only contains static JavaScript dedicated to this task and nothing else. Based on this, we consider it to be feasible that the code running in this origin is well audited and XSS-free. An HTML document hosted on this subdomain which contains all necessary scripts, is included in the main application's pages using an invisible iframe.

The main application communicates with the key handling scripts on the secure subdomain using the postMessage API [38]: The postMessage API is a mechanism by which two browser documents are able to communicate across domain boundaries in a secure manner. A postMessage can be sent by calling the method `postMessage(message, targetOrigin)`. While the `message` attribute takes a string message, the `targetOrigin` represents the origin of the receiving page. In order to receive such a message the receiving page hast to register an event handler function for the `message` event. When receiving a message via the event handler function, the browser passes additional metadata to the receiving page. This data includes the origin of the sender. Hence, the postMessage API can be used to verify the authenticity of the sending page.

After a successful key exchange, the component responsible for the initial handshake passes the session signing key via postMessage to the secure subdomain. The receiving script stores the key, depending on its configured lifespan, either via the subdomain's sessionStorage or localStorage mechanism [11].

4.2.4 JavaScript-Based Request Signing

Following the initial authentication, all further requests have to carry a correct HMAC signature to be recognized as authenticated. In consequence, all outgoing requests have to be initiated via JavaScript. This is done by replacing hyperlink targets and form actions with JavaScript event handlers, which pass the target URL to the signing component of our implementation. This component normalizes the request's data and then passes it, using the browser's post-message API, to the secure iframe (see Lst. 1).

As mentioned above, a central feature of the post-message API is, that the origin domain of the incoming requests is communicated in an unspoofable fashion. Hence, the request signing script can verify that the call to the signing function was created within an authenticated context (see Sec. 3.5),

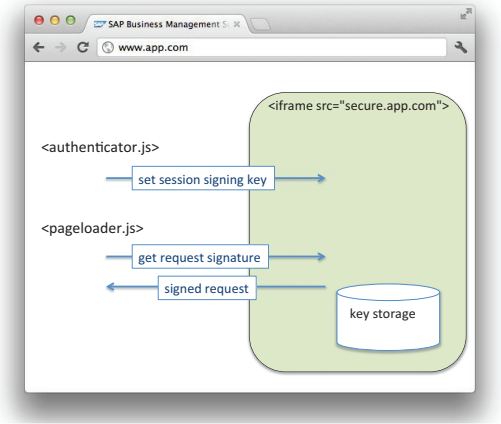


Figure 3: Domain isolated key handling

Code Listing 1 Request initation (simplified sketch)

```
<a href="#" onclick=
    "initAuthRequest('submod.jsp?a=b')">

<script>
window.addEventListener
    ("message", handleSignedRequest);
// Get request signature
function initAuthRequest(requestData){
    var rq = normalize(requestData);
    window.postMessage(rq,
                "http://secure.app.com")
   return false;
}
// receive signed request
function handleSignedRequest(event){
    if(event.origin ===
        "http://secure.app.com"){
 [attach request signature to request]
    }
}
</script>
```

and not by an untrusted third party which tries to abuse the functionality. Then, the signing component retrieves the signing key from localStorage, conducts the signing process, and passes the resulting values back to the main application, again using the postMessage functionality (see Lst. 2).

For apparent reasons, all page transitions and related request initiating actions of the main application have to utilize the request signing functionality. While for newly written applications, this won't cause a lot of effort, legacy applications have to be adapted to support the novel functionality. However, as discussed in [1], many applications can easily be adapted by traversing the application's pages DOM on load and patching the encountered links and forms to use the request signing functions. Alternatively, server-side rewriting of outgoing HTML could be utilized, modifying hyperlinks and form-actions to utilize JavaScript page navigation (see Lst. 1). Finally, for applications that mainly rely on AJAX driven client/server interaction, the request signing functionality can be introduced transparently replacing the XMLHttpRequest object with an object wrapper which implements the necessary actions.

Code Listing 2 Request signing code on the secure subdomain (simplified sketch)

```
window.addEventListener
        ("message", handleSignOrder);
// Create signed request
function handleSignOrder(event){
    if(authContext(event.origin)){
      var key = getSSK(event.origin)
      var sig = signReq(event.data, key)
      event.source.postMessage(sig,
                               event.origin)
    }
}
```

4.2.5 *Accessing Public Interfaces*

The final puzzle piece in the transitional implementation is a facility that enables external sites to navigate to the application's public interfaces (see Sec. 3.5). To recall, a public interface is a URL to which external sites are allowed to navigate in an authenticated state (e.g., for posting to social sharing sites).

For this purpose, we utilize a *pageloader* object: The page loader is a small JavaScript that is delivered by the application in case an unauthenticated request has been received for a URL which requires authentication and is contained in the application's set of public interfaces. The script is carried in the body of the initial 401 response during the key exchange handshake. In consequence, if such a response is received during a standard Web navigation process (opposed to the explicit authentication handshake executed by the native or transitional implementation), the page loader is executed in an otherwise blank HTML document.

The pageloader's source code is created dynamically by the server to contain the request's data which needs to be signed, in most cases mainly consisting of the original request's URL. The page loader dynamically includes the iframe to the secure subdomain and utilizes the standard request signing functionality of the implementation (see Lst. 1) to create a second, now authenticated request. The strict origin checking mechanism of the subdomain's signing interface robustly prevents potential abuse.

5. EVALUATION

5.1 Security Evaluation

In this section, we examine how capable BetterAuth is in mitigating the security threats (see Sec. 2).

Network-based attacks: At no point, passwords nor authentication tokens are transmitted over the network. Therefore, sniffing attacks are powerless. Also, due to the mutual authentication properties of the initial authentication, man-in-the-middle attacks are mitigated. However, please note, that BetterAuth only proves that the server indeed possesses the password. Furthermore, the security properties of BetterAuthdo not rely on the security of an underlying SSL/TSL connection. In consequence, SSL stripping attacks or CA breaches have no effect.

Issues related to cookie-based authentication tracking: There is no authentication cookie anymore, which could be stolen or manipulated. Hence, session hijacking and fixation attacks do not apply. Furthermore, CSRF attacks are

mitigated, as only *in-application* requests receive a signature, leading to a situation in which crossdomain requests are treated as unauthenticated by default. Only, if URLs are explicitly added to the list of public interfaces, the developer has to ensure, that crossdomain request to these URLs do not cause unwanted side effects. Finally, as we will discuss further in Sec. 5.3, Clickjacking attacks are partially addressed but still might occur.

Phishing: The password never leaves the browser. Hence, phishing attacks are bound to fail. However, this property only holds, if the password is entered only in BetterAuth-enabled input fields (see Sec. 5.3 for a further discussion of this limitation).

Limitations of the JavaScript implementation: Unlike a native implementation, the transitional implementation is susceptible to active man-in-the-middle attackers. The reason for this is, that the cryptographic components, which are executed in the secure subdomain's iframe are transported over the compromised network connection. Hence, the adversary could alter the transmitted source code in a fashion that leaks the session signing key or the user's password to the outside. Hence, at least the secure subdomain's content should be communicated via HTTPS.

5.2 Performance Evaluation

We don't expect a native implementation to cause considerable overhead. The utilized algorithms are in similar form already highly efficiently implemented both in browsers and servers as part of the SSL/TLS suite. Hence, the introduced overhead will be at most in the same range as overhead introduced by HTTPS communication. However, for the transitional implementation, the client-side component is implemented in pure JavaScript. Thus, a potential for noticeable overhead is given. Fortunately, in the last couple of years the browser vendors were inclined in an arms race on rapidly improving the performance of their JavaScript interpreters. To evaluate, how a JavaScript realization of the initial authentication would perform under realistic circumstances, we implemented the protocol as outlined in Sec. 4.2. For the cryptographic operations, we utilized the "Big Integer Library"[2] and the "Stanford Javascript Crypto Library (SJCL)"[3]. We benchmarked our implementation on three different machines running different operating systems each (Linux, Mac Os X, and Windows 7) and in total six browsers (see Tab. 2 for further details). The results of our benchmarking efforts can be obtained in Tab. 2. Among all configurations, the best performance could be observed with the Chrome browser, which reliably stayed below 300 ms, using a reasonable key length of 1024 Bit. The worst performance was exposed by Internet Explorer 9, which consumed in average 1314 ms for the same operations. Please keep in mind, that this overhead occurs only once during the whole process. The HMAC based authentication tracking can be implemented highly efficient and, thus, causes negligible performance effects.

5.3 Open Issues

The password entry field: BetterAuth provides strong protection against phishing attacks on the protocol level. However, this protection can be circumvented by the attacker on the GUI-level: As duly observed in [36, 12], if the user can be tricked into entering his password in non-BetterAuth form field, the attacker is still able to steal it. What is needed to close this hole is a visually unspoofable "trusted path" [43] from the user to a well isolated password handler within the browser. If such functionality is provided, further security guarantees in respect to the handling of the password data can be robustly introduced. Merely implementing such an approach is not a hard task on the engineering level, in parts it has already been done with the authentication dialogues for HTTP basic and digest authentication as well in various research prototypes. However, it is a major challenge in UI design. A right balance has to be found between the needs of Web UI designers and the ability of users to reliably recognize such "secure" entry forms.

Browser	D-H key length 768 Bit	1024 Bit	1536 Bit
Chromium/Linux[1]	116.7	261.1	876.6
Firefox/Linux[1]	182.6	426.8	1476.6
Chrome/Mac[2]	113.9	257.9	862.6
Safari/Mac[2]	405.6	942.5	3069.7
Chrome/Win 7[3]	127.2	281.9	932.7
IE 9/Win 7[3]	592.2	1314.6	4511.2

1: Acer Aspire, Ubuntu 11.10, Core i5, 2.53 GHz, 4GB RAM
2: MacBook Pro, Os X 10.7.2, Core i7, 2,2 GHz, 8GB RAM
3: ThinkPad T410s, Win7 Pro x64 SP1, Core i5 2,6 GHz, 4GB RAM

Table 2: JavaScript implementation performance (times in ms, averaged over ten runs)

Limited protection against Clickjacking: As motivated in Sec. 2.2.2, Clickjacking can be regarded as a class of vulnerabilities rooted in the current practice of authentication tracking. More precisely, Clickjacking is based on the adversary's capability to load cross-domain, authenticated GUI interfaces into the `iframe`. If BetterAuth is used without any configured public interfaces (see Sec. 3.5), this attack pattern would be infeasible, as no entry point into the application logic can be accessed from outside of an authenticated context. However, this protection ends as soon as public interfaces are added: In this case, the application probably offers a navigation path from the public interface to the targeted GUI interface. By tricking the user into multiple click-interactions with the disguised iframe, the attacker may be able to trick the user into unknowingly conduct this navigation. As all requests which originate from the public interface come from an authenticated context, they transparently receive request signatures, resulting in potential access of the attacker to the targeted Web GUI. Therefore, public interfaces should still be protected with anti-framing measures. Nonetheless, BetterAuth raises the bar of difficulty for Clickjacking attacks and with the set of public interfaces being limited and explicitly configured, full anti-framing protection is a straight forward task.

Replay attacks: If implemented in the from it is described in Sec. 3, the communication between browser and server would be susceptible to replay attacks from network attackers. We left handling of this issue out of the description for brevity and clarity reasons. However, adding replay protection to the authentication tracking process is straightforward, using a sliding window of monotonous growing nonces in the requests and limited state-keeping on the server-side.

[2] `http://leemon.com/crypto/BigInt.html`
[3] `http://crypto.stanford.edu/sjcl/`

6. RELATED WORK

Isolated security aspects of the Web authentication have received considerable attention, foremost the areas of phishing [8, 37, 41],cross-site scripting [25, 29, 20, 31]and CSRF [3, 33]. Due to the narrow focus of these works, we omit a detailled discussion. For the remainder of this section, we focus on password protocols and approaches that target Web authentication.

Password protocols: Bellovin and Merritt proposed the *Encrypted Key Exchange (EKE)* protocol that is based on pre-shared secretes, i.e. passwords, and secure against dictionary attacks [4, 5]. They put emphasis on considering which messages should be encrypted with the password without increasing the risk of offline, brute-force attacks. One drawback of this approach in modern web scenarios lies in the fact that the password has to be stored in cleartext on server side. Jablon proposed an improved approach that eliminates this need [18]. Wu proposed a modified version of EKE, called *Asymmetric Key Exchange (AKE)*, which is finally used to derive the *Secure Remote Password (SRP)* protocol [42]. It ceases to use symmetric cryptography and focuses on strong security properties with respect to leakage of server's user database or session keys. Steiner et al. describe the integration of a slightly modified version of Bellovin's and Merritt's approach [4], named *DH-EKE*, into TLS [40]. This way, they eliminate the need for a public key infrastructure. Due to mutual authentication, certificates become obsolete. Their *Secure Password-Based Cipher Suite for TLS* implements confidentiality and authenticity.

Web authentication: *SessionLock* [1] is closely related to our transitional JavaScript implementation of BetterAuth. The paper demonstrates how standard JavaScript can be used to substitute cookie-based authentication tracking with a browser-driven HMAC scheme. SessionLock does not protect against CSRF problems and does not handle the initial authentication step. [2] introduces browser authentication without user authentication: The browser, generates a self-signed certificate. This certificate must not contain any user-related information. A new certificate is issued by the client for every single server domain ("origin"). Session tracking can be secured by relating session cookies to the respective client certificate, hence, mitigating several of the cookie related threats, such as SID theft or session fixation. Finally, the most recent draft of HTTP/1.1 specification provides an extension of long-known HTTP Basic and Digest Authentication based on Challenge-Response Authentication [13], which prevents known eavesdropping attacks on the former HTTP authentication standards.

Chen et al. address the problem of cross-site attacks that occur while surfing sensitive and non-trustworthy websites at the same time in one browser [7]. Therefore, they isolate browser sessions which prevents cross-domain attacks. Same-domain attacks are out of scope of this approach. This feature is comparable with our context dependent authentication and public interfaces. The security level of *app isolation* is equivalent to surfing different websites using different browsers.

7. CONCLUSION

In this paper we presented BetterAuth. BetterAuth is a mutual Web authentication protocol that was designed to be secure by default, thus, freeing the developers and operators of Web applications from the need to counter potential threats at various heterogeneous places in the application's architecture, as it is required by the currently established approach. BetterAuth significantly improves the susceptibility of the authentication process to known threats, ranging from network attacks, over Cross-site Request Forgery, up to Phishing. Furthermore, the protocol was designed in a fashion that allows an implementation in standard JavaScript, enabling its deployment even in situation in which no widespread native browser support is present yet.

8. REFERENCES

[1] B. Adida. Sessionlock: securing web sessions against eavesdropping. In *Proceeding of the 17th international conference on World Wide Web*, WWW '08, pages 517–524, New York, NY, USA, 2008. ACM.

[2] D. Balfanz, D. Smetters, M. Upadhyay, and A. Barth. TLS Origin-Bound Certificates. [IETF draft], `http://tools.ietf.org/html/draft-balfanz-tls-obc-01`, Version 01, November 2011.

[3] A. Barth, C. Jackson, and J. C. Mitchell. Robust Defenses for Cross-Site Request Forgery. In *CCS'09*, 2009.

[4] S. M. Bellovin and M. Merritt. Encrypted Key Exchange: Password-Based Protocols Secure Against Dictionary Attacks. In *Proc. IEEE Computer Society Symposium on Research in Security and Privacy*, pages 72–84, Oakland, CA, May 1992.

[5] S. M. Bellovin and M. Merritt. Augmented Encrypted Key Exchange. In *Proceedings of the First ACM Conference on Computer and Communications Security*, pages 244–250, Fairfax, VA, November 1993.

[6] A. Bortz, A. Barth, and A. Czeskis. Origin Cookies: Session Integrity for Web Applications. In *W2SP 2011*, 2011.

[7] E. Y. Chen, J. Bau, C. Reis, A. Barth, and C. Jackson. App Isolation: Get the Security of Multiple Browsers with Just One. In *18th ACM Conference on Computer and Communications Security (CCS)*, 2011.

[8] R. Dhamija and J. Tygar. The Battle Against Phishing: Dynamic Security Skins. In *Symposium On Usable Privacy and Security (SOUPS) 2005*, July 2005.

[9] P. Eckersley. How secure is HTTPS today? How often is it attacked? [online], `https://www.eff.org/deeplinks/2011/10/how-secure-https-today`, October 2011.

[10] P. Eckersley and J. Burns. The (Decentralized) SSL Observatory. Invited Talk, Usenix Security 2011, `http://static.usenix.org/events/sec11/tech/slides/eckersley.pdf`, August 2011.

[11] I. H. (Ed. Web Storage. W3C Candidate Recommendation, `http://www.w3.org/TR/webstorage/`, December 2011.

[12] J. Engler, C. Karlof, E. Shi, and D. Song. Is it too late for PAKE? In *Proceedings of W2SP*, 2009.

[13] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, Y. Lafon, and

J. Reschke. HTTP/1.1, part 7: Authentication. [IETF draft], http://tools.ietf.org/html/draft-ietf-httpbis-p7-auth-18, Version 18, January 2012.

[14] Google. Safe Browsing for Firefox. [application], http://www.google.com/tools/firefox/safebrowsing/, (03/20/06), 2006.

[15] R. Hansen and J. Grossman. Clickjacking. [online], http://www.sectheory.com/clickjacking.htm, last accessed 02/13/12, August 2008.

[16] E. Henning. Trustwave issued a man-in-the-middle certificate. [online], http://www.h-online.com/security/news/item/Trustwave-issued-a-man-in-the-middle-certificate- 1429982.html, February 2012.

[17] J. Hodges, C. Jackson, and A. Barth. HTTP Strict Transport Security (HSTS). [IETF draft], http://tools.ietf.org/html/draft-ietf-websec-strict-transport-sec-04, Version 04, January 2012.

[18] D. Jablon. Extended Password Key Exchange Protocols Immune to Dictionary Attacks. *Enabling Technologies, IEEE International Workshops on*, 0:0248, 1997.

[19] C. Jackson and A. Barth. ForceHTTPS: Protecting High-Security Web Sites from Network Attacks. In *WWW 2008*, 2008.

[20] M. Johns. SessionSafe: Implementing XSS Immune Session Handling. In *European Symposium on Research in Computer Security (ESORICS 2006)*. Springer, September 2006.

[21] M. Johns, B. Braun, M. Schrank, and J. Posegga. Reliable Protection Against Session Fixation Attacks. In *26th ACM Symposium on Applied Computing (SAC 2011)*. ACM, March 2011.

[22] D. Kaminsky. h0h0h0h0. Talk at the ToorCon Seattle Conference, http://seattle.toorcon.org/2008/conference.php?id=42, April 2008.

[23] A. Klein. "Divide and Conquer" - HTTP Response Splitting, Web Cache Poisoning Attacks, and Related Topics. Whitepaper, Sanctum Inc., http://packetstormsecurity.org/papers/general/whitepaper_httpresponse.pdf, March 2004.

[24] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, http://tools.ietf.org/html/rfc2104, February 1997.

[25] M. T. Louw and V. Venkatakrishnan. BluePrint: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers. In *IEEE Symposium on Security and Privacy (Oakland'09)*, May 2009.

[26] M. Marlinspike. New Tricks For Defeating SSL In Practice. Talk at the Black Hat DC conference, 2009.

[27] Microsoft. Ie8 security part vii: Clickjacking defenses, 2009.

[28] MSDN. Mitigating Cross-site Scripting With HTTP-only Cookies. [online], http://msdn.microsoft.com/workshop/author/dhtml/httponly_cookies.asp, (01/23/06).

[29] Y. Nadji, P. Saxena, and D. Song. Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In *Network & Distributed System Security Symposium (NDSS 2009)*, 2009.

[30] J. Nelson and D. Jeske. Limits to Anti Phishing. In *Proceedings of the W3C Security and Usability Workshop*, 2006.

[31] N. Nikiforakis, W. Meert, Y. Younan, M. Johns, and W. Joosen. SessionShield: Lightweight Protection against Session Hijacking. In *3rd International Symposium on Engineering Secure Software and Systems (ESSoS '11)*, LNCS. Springer, February 2011.

[32] Y. Oiwa, H. Watanabe, H. Takagi, B. Kihara, T. Hayashi, and Y. Ioku. Mutual Authentication Protocol for HTTP. [IETF draft], http://tools.ietf.org/html/draft-oiwa-http-mutualauth-10, Version 10, October 2011.

[33] Open Web Application Security Project. Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet. [online], https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet, accessed November 2011, 2010.

[34] J. Ruderman. The Same Origin Policy. [online], http://www.mozilla.org/projects/security/components/same-origin.html (01/10/06), August 2001.

[35] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson. Busting Frame Busting: a Study of Clickjacking Vulnerabilities on Popular Sites. In *Web 2.0 Security and Privacy (W2SP 2010)*, 2010.

[36] D. Sandler and D. S. Wallach. <input type="password"> must die! In *Web 2.0 Security and Privacy (W2SP)*. IEEE, May 2008.

[37] M. Sharifi, A. Saberi, M. Vahidi, and M. Zoroufi. A Zero Knowledge Password Proof Mutual Authentication Technique Against Real-Time Phishing Attacks. In P. D. McDaniel and S. K. Gupta, editors, *ICISS*, volume 4812 of *Lecture Notes in Computer Science*, pages 254–258. Springer, 2007.

[38] E. Shepherd. window.postmessage. [online], https://developer.mozilla.org/en/DOM/window.postMessage, last accessed 02/12/12, October 2011.

[39] S. Stamm, Z. Ramzan, and M. Jakobsson. Drive-by Pharming. In *In Proceedings of Information and Communications Security (ICICS '07)*, number 4861 in LNCS, December 2007.

[40] M. Steiner, P. Buhler, T. Eirich, and M. Waidner. Secure Password-Based Cipher Suite for TLS. In *NDSS*, pages 134–157, 2001.

[41] M. Wu, R. C. Miller, and G. Little. Web Wallet: Preventing Phishing Attacks by Revealing User Intentions. In *Proceedings of the second symposium on Usable privacy and security (SOUPS 06)*, 2006.

[42] T. Wu. The secure remote password protocol. In *Proceedings of the 1998 Internet Society Network and Distributed System Security Symposium*, pages 97–111, 1998.

[43] K.-P. Yee. User interaction design for secure systems. In *Proceedings of the 4th International Conference on Information and Communications Security*, ICICS '02, pages 278–290, London, UK, UK, 2002. Springer-Verlag.

Using Memory Management to Detect and Extract Illegitimate Code for Malware Analysis

Carsten Willems
Ruhr-University Bochum,
Horst Görtz Institute for
IT-Security (HGI), Germany

Felix C. Freiling
Friedrich-Alexander University
of Erlangen-Nuremberg,
Germany

Thorsten Holz
Ruhr-University Bochum,
Horst Görtz Institute for
IT-Security (HGI), Germany

ABSTRACT

Exploits that successfully attack computers are typically based on some form of shellcode, i.e., illegitimate code that is injected by the attacker to take control of the system. Detecting and gathering such code is the first step to its detailed analysis. The amount and sophistication of modern malware calls for automated mechanisms that perform such detection and extraction.

In this paper, we present a novel generic and fully automatic approach to detect the execution of illegitimate code and extract such code upon detection. The basic idea is to flag certain memory pages as non-executable and utilize a modified page fault handler to react on the attempt to execute code from them. Our modified page fault handler detects if legitimate code is about to be executed or if the code originates from an untrusted location. In such a case, the corresponding memory content is extracted and execution is continued to retrieve more illegitimate code for analysis.

We present an implementation of the approach for the Windows platform called *CWXDetector*, which involved reverse-engineering the proprietary memory management system of this operating system. Evaluation results using a large corpus of malicious PDF documents show that our system produces no false positives and has a very low false negative rate. To further demonstrate the universality of our approach, we also used it to detect shellcode execution in *Flash Player*, *RealVNC client*, and *VideoLan Client*.

1. INTRODUCTION

No matter what particular exploitation method or target is used, the ultimate aim of an attacker is to perform *malicious computation* on the target system, i.e., to execute machine instructions whose type and order are under the complete control of the attacker. Usually, malicious computation is conducted using *illegitimate code*, i.e., code that was not intended to be executed, neither by the developer of the exploited software nor by the end-user of the system. Such code is usually injected into the target system using external data like network traffic or application files.

As a countermeasure to this threat, operating systems try to *prevent* the execution of illegitimate code using techniques like *Data Execution Prevention* (DEP) [17] and *Address Space Layout Randomization* (ASLR) [33]. However, prevention alone does not help in the process of analyzing malicious code since we also want to analyze what an attacker attempts to do when compromising a system. Therefore, it is necessary to develop mechanisms that *detect* and *extract* illegitimate code from malicious data. A consecutive analysis of the extracted data then can assist in developing new protection techniques and creating signatures for zero-day malware until patches are available.

In this work, we present an approach to automatically detect and extract illegitimate code by instrumenting the memory management features of operating systems. Roughly speaking, the idea is to mark certain memory pages as non-executable. This ensures that upon execution of code in these regions the page fault handler of the operating system is called. This usually suffices to *detect* illegitimate code. However, to *extract* illegitimate code, we modify the page fault handler so that the memory page that caused the page fault is written to a dump file for later analysis.

The power of our approach stems from its simplicity in using the page fault handler of the operating system itself. Therefore, the challenge is to evaluate its effectiveness in practice. This means to evaluate it for the Windows platform, since this is still the major target of illegitimate code today. To that end we have implemented *CWXDetector*, a new tool for the analysis of malware on Windows. Unfortunately, Windows is a closed-source operating system and without proper documentation it is a difficult task to integrate custom functionality into the kernel. Therefore, we had to perform a lot of substantial reverse engineering regarding the internal memory management mechanisms of the Windows kernel. Due to space constraints, most findings of this research are covered in an accompanying technical report [35]. Additionally, due to space limitations we have published an extended version of this document in a second technical report [36]. This technical report covers more implementation details of the tool *CWXDetector* and comprehensive analysis results.

We evaluated our approach by considering the task of detecting and extracting illegitimate code in/from a particularly relevant class of application files, namely those in Adobe's Portable Document Format (PDF) [22]. Our approach proves to be very effective. We analyzed a set of 7,278

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

malicious PDF documents using a set of vulnerable versions of Adobe Reader and achieved a detection rate of 93.2%. This can be regarded as a lower bound for our method since many of the investigated PDF files appeared to be broken although they were flagged as malicious by Antivirus products. We also analyzed the same amount of benign PDF documents, resulting in a (false positive) detection rate of 0%. Furthermore, our detection results compare favorably to those of application specific detection tools like *Wepawet* [6], *pdf examiner* [34] and *ADSandbox* [7], but outperforms them by being generic and – to some extent – also being capable of detecting zero-day exploits. To further demonstrate the universality of our approach, we also used it to detect shellcode execution in *Flash Player*, *RealVNC client*, and *VideoLan Client*. Especially the Flash file was complicated since it consists of a Flash file embedded in a Word document, which is an attack vector that is hard to analyze with concurrent approaches like ShellOS [27].

In summary, the contributions of this work are as follows:

- We present a generic and fully automatic analysis approach to detect the execution of illegitimate code and extract such code upon detection.
- We successfully evaluate our approach using malicious PDF documents as example and show that we can improve state-of-the-art tools. In addition, our approach can also be applied to other kinds of malicious documents as we demonstrate in three case studies.
- Furthermore, we have reverse engineered essential parts of the memory management internals of the Windows operating system. Due to space constraints, we only cover the relevant aspects in this paper but full details are available in an additional technical report [35].

2. RELATED WORK

There has been a lot of work on shellcode analysis in the past and we review work that is closely related to ours. Note that our approach is not meant to *protect* a system, but to monitor and analyze the execution of illegitimate code in an analysis environment. To that end, our system even disables some security measures like DEP. Nevertheless, there are some similarities in other preventive and analysis techniques that we now discuss.

2.1 Preventive Measures

A large body of related work mainly aims at the *prevention* of malicious code execution, mostly following the *reference monitor* approach. Many such methods are directly integrated into contemporary compilers and operating system [35]. However, often newly introduced protection techniques are incompatible to existing old applications and, therefore, can be disabled by the applications themselves or are deactivated per default. *Microsoft's EMET tool* [16] tries to overcome this problem by allowing a process-specific configuration of these protection methods and their enforcement. Other methods restrict memory write operations or control transfers. Kiriansky, Bruening and Amarasinghe [13] as well as Abadi et al. [1] use code rewriting techniques to implement their restrictions. Instead of modifying parts of the operating system, they inline checks into the executed applications. Unfortunately, they mostly lack the capability of handling self-modifying and dynamically created code and they are unable to detect control flow transitions that occur due to *structured exception handling* and not due to a regular branch instruction. Another difference to our work is that these solutions terminate the protected process in the event of a security violation, and are not able to produce any further analysis data. Seshadri et al. propose *SecVisor*, a tool that is capable of ensuring code and execution integrity by utilizing a tiny hypervisor [23]. In contrast to our approach, they aim at kernel code and present a solution for Linux systems.

To some extent our approach is similar to DEP [17], which also employs the *no-execute/execute-disable* (NX) flags of the page table entries to block the execution of certain memory regions. Nevertheless, there are significant differences to our system, since we are aiming at the analysis of malicious code and not at preventing its execution. Therefore, we intentionally permit the execution of illegitimate code after extracting it. Furthermore, we do not only consider the type of memory when deciding which should be monitored respectively executed, but we also check the initiator of memory related modifications and allocations. Additionally, we restrict the unintended allocation of executable memory due to programming errors, which would not be detected by DEP and enables us to interact if code is copied into and executed within those memory regions.

In summary, all of the previously described measures aim at the prevention of malware execution, but offer no assistance in their further analysis.

2.2 Detection of Illegitimate Code

The detection of illegitimate code is an extremely difficult problem today. Early attempts relied on static signatures [11], but had to be improved due to the heavy use of polymorphism, encryption and other obfuscation methods. More enhanced methods try to detect certain invariant parts of the shellcode, e.g., Akritidis et al. [2] search for the typical "sled component" in such code. Others have used heuristics in combination with *dynamic* analysis methods to detect illegitimate code. For example, machine learning methods have been used to deal with the variable parts, e.g., Payer, Teufl and Lamberger utilize a neural network [18] in combination with *execution chain evaluation*. Polychronakis, Anagnostakis and Markatos [19] use emulation to detect an ongoing decryption process which is typical for polymorphic shellcode. Also Baecher and Koetter [3] use an emulated environment to identify and isolate shellcode with the help of *GetPC heuristics*. In order to overcome the drawbacks of such emulators, Snow et al. use a KVM hypervisor to execute instruction sequences that are found in network streams or arbitrary buffer content [27]. By applying sophisticated runtime heuristics they are able to detect malicious code. Overall, and in contrast to *CWXDetector*, these signature- and heuristics-based approaches are not fully generic and have to be extended when new anti-detection measures of malicious code come up. Additional hypervisor-assisted approaches to detect and analyze the execution of malicious code have been presented by Dinaburg et al. [8] as well as Litty, Lagar-Cavilla, and Lie [14]. Different to ours these systems do not modify the operating system itself, but introspect solely from the *outside*, i.e., the hypervisor. It has been observed before that the memory system itself can be used to detect illegitimate code execution. For example, the *PaX project* [32] proposes several different measures to implement non-executable memory — even on architectures

with no hardware support for that. Similar detection mechanisms can obviously be realized with hardware-DEP, like explained in the previous subsection.

2.3 Extraction of Illegitimate Code

Several solutions aiming at the extraction of illegitimate code exist, especially for automated unpacking of malware. These mechanisms usually interact deeply with the memory management of the underlying operating system and try to detect the execution of memory regions which have been written to beforehand. *OllyBone* [31] implements this by instrumenting the *translation lookaside buffers* [10]. Since *OllyBone* is a debugger-plugin, it imposes all the disadvantages of debugger-driven malware analysis, e.g., its detectability. Another disadvantage that contrasts it to our approach is that it is a semi-automated process, in which execution is stopped at the first occurrence of malicious instructions and the human analyst has to continue with further extraction steps. Finally, it is not able to deal with dynamically allocated memory regions (since it focuses on Windows PE sections).

OmniUnpack [15] uses an approach similar to the *PAGEEXEC* method proposed by PaX, i.e., the *User/Supervisor* page table flag is used to automatically break on the execution of certain monitored pages. In order to decide whether executed and previously written memory should be considered as malicious, an external detector is used to scan memory after unpacking for the existence of malicious code. That detector, again, has to use signatures or heuristics that generate a lot of false positives, especially if a JIT-compiler is involved. Finally, executed memory is only considered if a critical system call is executed afterwards. Our approach uses a more effective approach, based on the concept of *trusted callers*, that results in much better detection results.

Renovo [12] runs the sample within the emulated environment TEMU [28] and maintains shadow memory to track written memory regions. Since this cannot be done on a native system, it cannot be realized *without* system-emulation. Again, like with debuggers, this enables the monitored malware to detect the synthetic environment.

A similar approach to ours has been published by Porst in form of a *Pin-tool* [20]. It uses a simple heuristic that identifies and dumps shellcode, if its instructions are not located in the code section of any loaded module, but on the stack or the heap.

3. MODEL AND DEFINITIONS

We now specify our attacker model, define the term *illegitimate code*, and further concretize our two aims: the detection and extraction of executed illegitimate code.

3.1 Attacker Model

In this work we assume a remote attacker that provides some malicious piece of data in order to exploit a vulnerability in some handling application resulting in the execution of shellcode. This data may have arbitrary form, e.g., a specially crafted PDF document, a Flash file, or a malicious input packet to some network application.

We are aware of the threats posed by *return oriented programming* [24] (ROP) or JIT-spraying [4, 25] techniques, but nevertheless assume that an attack does not *fully* consist of such code. To the best of our knowledge, we are not aware of any incidents in the past that used single staged full-ROP/JIT-sprayed attacks.

3.2 Illegitimate Code

Our approach enforces the partitioning of executable memory into regions that contain legitimate code and those that may contain illegitimate one. Additionally, it monitors and reacts on the execution of instructions that are located in illegitimate code regions. Intuitively, all code that belongs to the operating system or to a known application is legitimate. For realizing this distinction, we first partition the files of a system into a set of trusted files and a set of untrusted ones. We assume that such a distinction is given, e.g., by defining all files in a freshly installed system as trusted. For dealing with dynamically created code, we identify those code portions contained in trusted files that should be allowed to allocate executable memory. Accordingly, we partition each trusted file into trusted memory modification functions and untrusted ones. Again we assume such a distinction is given, e.g., by defining all required code emitting memory functions of the operating system as trusted and all others as untrusted. Then, *legitimate code* (LC) is code which is either contained in a trusted (system or application) file or that was dynamically created by any of the trusted functions from one of those files.

We define *illegitimate code* (ILC) as code that is not legitimate. Intuitively, ILC is code which would not be executed if the operating system and the installed applications would function properly. In practice it is code that is either injected by or constructed on behalf of an attacker by some malicious piece of software or data. Therefore, ILC is similar to *shellcode* in its current understanding.

3.3 Problem Statement

The aim of the system described in this work is to perform two tasks in an automated way:

1. *Detect* the execution of illegitimate code, and
2. *extract* that code, i.e., dump all relevant memory pages to a file, for a later in-depth analysis.

4. APPROACH: INSTRUMENTING THE PAGE FAULT HANDLER

In the following we describe our approach and sketch our Windows implementation of it. More details are available in two accompanying technical reports [35, 36].

4.1 Enforcing an Invariant

Based on our attacker model, no matter what kind of exploit is used in an attack, the resulting effect is always the execution of illegitimate code like we have defined above. To that end, when a vulnerability is exploited, the control flow is redirected to some code hosting location on the stack, the heap, or in a static data area.

Throughout our approach, we establish and maintain the following invariant condition: *all ILC resides in* non-executable *memory*. As an effect to this invariant, all execution attempts of ILC will result in the invocation of the *page fault* handler of the operating system. By implementing our own custom page fault handler, we are able to react on such attempts appropriately.

4.2 Trusted Files and Functions

To establish the invariant, we need to identify the set of trusted files and functions. For simplicity we trust all files which have been already existing when we start our analysis, and distrust all files which were created or modified during later system operation. To achieve this, we need to keep track of file manipulation operations. Therefore it is necessary to intercept ("hook") calls of system services that are used to create or open files with write access.

For each trusted file we further define a set of *trusted memory modification functions*, which contains all the functions that are allowed to dynamically create executable memory or modify the protection settings of already existing memory to being executable. The set of all such functions from all trusted files is called *trusted callers*.

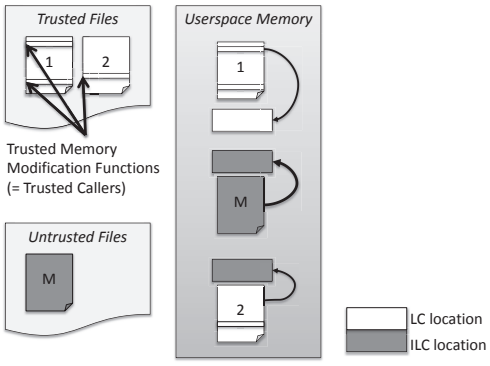


Figure 1: Trusted Memory Modification Functions

Figure 1 illustrates our understanding of trust, showing an example with two trusted and one untrusted file (left side of the figure). While trusted file 1 contains two trusted memory modification functions, trusted file 2 only has one. Obviously, the untrusted file can not contain any trusted function at all. The simplified version of the userspace memory (right side of the figure) shows that all three files have been mapped into the virtual address space. The memory related to the trusted files only contains trusted code, hence constitutes LC memory, whereas that memory of the untrusted file may contain illegitimate code. Furthermore, each mapped module has allocated one dynamic memory area, pointed to by the corresponding arrow. That area belonging to file 1 was created by a trusted caller and, hence, may contain legitimate executable code. However, the memory created by the untrusted caller from file 2 as well as the region created by the untrusted file may contain ILC and, therefore, are marked as ILC locations.

4.3 Memory Protection Modifications

Obviously it is necessary to intercept attempts to modify the memory protection, because this can result in executable memory. This is realized by hooking critical system calls. Inside these hook functions we enforce the following properties to maintain our invariant:

- only trusted callers can allocate executable memory,
- only trusted callers can modify existing memory to being executable, and
- only trusted files can be mapped into executable memory.

The parameters of all system calls that violate these rules are manipulated transparently such that the resulting memory regions become *non-executable*. In summary, only trusted files can be loaded into executable memory and only trusted callers can create executable memory. There is one exception: even if a *trusted caller* tries to modify the memory protection, intervention may be necessary under special circumstances: if the related target memory belongs to a mapped trusted file and should become writable, the executable right has to be removed. This enforces the $W \oplus X$ property [17]. In general, all legal linkers should produce files which fulfill this requirement anyway. Nevertheless, we enforce it on our own to also handle files securely which violate it by intent or accident.

The realization of making memory non-executable strongly depends on the underlying system architecture and also on the operating system. Many contemporary CPUs offer an NX protection flag on a page level. Nevertheless, this feature can be only used for valid page table entries (PTEs) and, therefore, in most cases some additional OS memory objects may have to be modified as well (more details are available in the technical report [36]).

4.4 Custom Page Fault Handler

The heart of our detection method is the *custom page fault handler*, which reacts on the attempt to execute memory regions which we have marked non-executable beforehand. As described above, all necessary prerequisites are already done when new memory is allocated or the protection of already existing one is tried to be modified. Accordingly, the custom page fault handler only has to *wait* until a protection-related page fault is triggered. In that event, we have to check if the occurred page fault is really related to our system modifications. If so, we have detected the execution of illegitimate code and, as a result, copy the content of the related memory page to a dump file. Furthermore, we log the current instruction pointer to indicate the particular instruction that should be executed within the dumped page. After that, the related memory region is modified by us to being executable, such that the current and all further execution attempts for this page will become successful. This is done because we do not want to stop our analysis process once the first illegitimate instruction is found. Finally, we resume the current process and wait for the next fault.

4.5 Multi Version Dumping

In order to avoid detection, shellcode is very often built by multiple stages that are organized like a russian stacking doll (*matrushka*). Each stage is only a small stub that deobfuscates or decrypts the next stage and then transfers control to it. Since the effective malicious instructions are mostly contained in the final stage, it is desirable to unpack it automatically. Therefore, we have developed an additional feature called *multi version dumping* (MVD), in which different versions of each executed page may be created. To that end, an internal copy of each dumped memory page content is stored and if the content is modified later on, another dump file is created. By comparing two consecutively created dumps, we can isolate those parts that have been modified and infer the decrypted shellcode instantly.

Notice, that not every shellcode is multi-staged. Therefore, sometimes only one dump file is created for a detected ILC containing memory page, and sometimes two or more.

If multiple versions are created, we mostly are interested in the final one, since we assume that it contains the fully decrypted code. However, also considering the intermediary stages may be reasonable, i.e., to gain knowledge about the used unpacking method.

4.6 Windows-based Implementation

We have developed the tool *CWXDetector* as a concrete implementation of our approach for the x86 version of Windows XP. One reason for choosing this particular version was the availability of a large sampleset of malicious documents for that target OS. Nevertheless, our system could easily been migrated to (at least the 32bit versions of) Windows Vista or 7. Due to space constraints, a detailed explanation of this tool and its internals can be found in the accompanying technical report [36]. In the following, we only provide a brief overview of the involved tasks and difficulties. As OS platform we have utilized the PAE kernel version of Windows, since this one supports the NX page table flag that we need to realize non-executable memory. Although that kernel version was originally intended to support physical memory that is larger than 4 GB, it is nowadays used on all installations of the 32 bit Windows XP version that have DEP enabled. To realize our approach we had to

- define trusted files and trusted callers,
- implement hook functions for memory allocations and protection modifications,
- implement a custom page fault handler to handle ILC execution, and
- additionally modify essential system functions to support our approach.

Since Windows is not an open source operating system, a lot of reverse engineering had to be performed previously, especially on the underlying memory objects like VADs and PPTEs. The detailed findings of this work are explained in the second accompanying technical report [35]. Though the implementation of *CWXDetector* seems to be straightforward, the unavailability of the Windows source code posed enormous difficulties when intercepting kernel system calls, customizing the page fault handler, and dynamically patching OS-controlled memory management resources.

5. ANALYSIS OF PDF DOCUMENTS

For illegitimate code detection and extraction, *CWXDetector* has to perform a dynamic analysis and actually parse and process the malicious content with the intended client application. To further illustrate and evaluate its effectiveness, we apply it to the analysis of PDF documents. Malicious documents as attacking vector have become very popular in the past years, and especially the PDF format is a commonly used medium for malicious content. One reason for that is its extensive feature list: the programming languages *Javascript* and *Actionscript* can be used within and many different object types like images, sounds and even executables can be embedded. Accordingly, the underlying codebase is very complex and hence error-prone. For example, the latest PDF reference [9] contains 756 pages and *Adobe* already has published several extensions to it.

Since dynamic analysis in general is *incomplete*, we test each PDF sample in different viewer applications and then combine all the findings. Though not usual, a malicious functionality may only be triggered on a certain user action or input. For correctly analyzing also those files, user-simulation could be employed as an extension to the existing functionality.

We set up multiple virtual machines with 32 bit Windows XP SP2 and installed a different PDF viewer application on each of them. In particular, we used *Adobe Acrobat Reader* 6.0.0, 7.0.0, 7.0.7, 8.1.1, 8.1.2, 8.1.6, 9.0.0, 9.2.0 and 9.3.0. For comparison we also set up one machine with *Foxit Reader* 3.0.0 for which also some vulnerabilities are known. This particular application and version set was chosen to cover the most of the known vulnerabilities for PDF documents, but it should be mentioned that it may not be optimal nor have full coverage for all known existing exploits. Each PDF sample was then analyzed in all of those machines in parallel. During the analysis we performed the following steps on each machine separately:

- We installed the customized page fault handler and the system hooks.
- We disabled DEP for the viewer application, since otherwise the execution attempt of non-executable code would crash the process and we would not have any possibility to intercept it.
- We opened the document in the viewer application.
- If new memory was allocated or existing memory was modified during execution, we enforced the invariant from Section 4.1.
- If the execution of illegitimate code was detected, we dumped the associated memory page to a file and modified the related PTE to being executable. We then checked the dumped memory page for typical patterns of illegal code [36]. In case such a pattern was found, we labeled the execution with *PATTERN*.
- If a new process was created by the PDF viewer, we marked the execution with *PROCESS*. We prevented the spawning of additional processes since we are only interested in analyzing exploits in the PDF viewer application itself.
- If a dialog window was shown by the PDF viewer, we assigned the label *DIALOG* and additionally logged the contents of the window. We then simulated a user input to close the window and continued viewing the PDF document.

For scalability reasons the analysis process was stopped after a specific timeout, which was set to two minutes in our experiments. As almost all known malicious PDFs trigger their malicious operations instantly when viewing the first page of the document, it is safe to assume that we trigger most of all malicious shellcodes after this short amount of time. In many cases the viewing application was terminated prematurely, long before that timeout was reached. In that case we marked the execution as *CRASH*. Finally, and if none of the aforementioned labels have been assigned, the case was labeled as *NOTHING*.

Overall, every PDF file ended up with a combination of two labels (d, c): the first label d determined whether illegal code execution was detected or not, and the second label c was either *PATTERN*, *CRASH*, *PROCESS*, *DIALOG*, *NOTHING* as defined above. Since different PDF viewers can react differently to a single PDF file, we needed to aggregate all the different results into one overall value. We defined a lexicographic total order on the tuples as follows: $(d, c) > (d', c')$ if and only if either d had detected illegal code and d' not, or (if $d = d'$) $c > c'$ according to the following ordering:

$$PATTERN > CRASH > PROCESS > DIALOG > NOTHING$$

As the final result, we used the highest occurring value as combined overall value.

As stated above, we further have to determine the set of trusted files and trusted callers. Since we start with an uninfected clean system, it is safe to trust all files that already exist when the analysis starts and distrust all modified and created ones. The trusted callers are determined in a semi-automated way: we start with a clean whitelist and each time executable memory is generated, we manually inspect the call stack and verify if the caller is legitimate. This only has to be done once for a given use case, for instance with the help of a known benign PDF document. For this particular experiment we came up with two functions (`LdrpSnapIAT` and `LdrpSetProtection`) in the *Windows Native Library (ntdll.dll)* and one in the JIT-compiler of *Acrobat Reader (authplay.dll)*. It is obvious that this incremental approach is fail safe, since we only can get *false positives* if we have forgotten particular trusted callers, but will never create *false negatives* if we set up our trusted caller list correctly.

In Section 6 and 7 we describe our findings and the results of our experiments in detail. Mostly we are interested in the information if a viewed malicious PDF document triggers the execution of ILC or not. If we are able to detect such an attempt, we call our result a *true positive*. If we fail to detect it, we call it a *false negative*. If on the other hand, a benign PDF document is analyzed and in reality no ILC is executed at all, but our system erroneously reports ILC execution, we call this a *false positive*. Finally, a *true negative* stands for such a case, in which our system correctly does not report ILC execution.

6. DETECTION EVALUATION

We have evaluated the detection quality of our system by means of two different experiments. First, we have performed a comprehensive analysis of PDF documents. For that purpose we have created two sets of PDF documents of size 7,278 each (a *benign* set and a *malicious* set) and used *CWXDetector* to analyze these files. We have developed heuristics to measure the correctness and completeness of our findings. We further have compared our generic system against several other analyzers, that use application-specific knowledge to analyze PDF documents. In a second experiment we briefly illustrate the universality of our approach by applying our tool on malicious *Flash* documents and network packets.

6.1 Benign PDF Sampleset

In order to test the *false positive* rate of our approach, we obtained a set of known benign documents from the TOP 5000 Internet sites from `www.alexa.com`. We have then selected those files that contain as much different PDF features as possible. Accordingly, analyzing them enables us to monitor various different behaviors, in form of code coverage of the PDF viewing application and its plugins. The resulting set has the following characteristic: altogether it contains 7,278 samples, 600 of which contain *Javascript*, 782 contain *AcroForms*, 1,573 samples have an *OpenAction*, and 751 some *AdditionalAction*. All samples have been verified by the publicly available AV service Virus Total [26] and in 3 cases one or more supported scanners returned a positive result. We checked those samples by hand and did not find any malicious content within them. Therefore, most probably these detections are AV false positives.

We ran our system on the benign sample set. As a result, we did not detect *any* single ILC execution, resulting in a *false positive* rate of 0% for this particular set. To speed up the analysis, we only used the three "most vulnerable" PDF viewer applications for the benign sample set (namely *Adobe Acrobat Reader 7.0.7, 8.1.1* and *9.0.0*). Due to the achieved false positive rate of our experiments of zero, we can assume that it will not increase dramatically by using more different viewers.

6.2 Malicious PDF Sampleset

We obtained a set of 7,278 known malicious PDF documents from a well-known AV vendor. The set consisted of all their valid incoming PDF samples from January 2011. These samples originated from different sources: *70.0%* from sample sharing between AV vendors, *24.0%* found in the wild, *4.8%* from multiscanner projects and *1.2%* from intercepted botnet traffic. We checked all samples with Virus Total [26] which confirmed that all of them were indeed malicious. We ran our tool on the malicious sample set and were able to detect and extract ILC in 93.2% of all cases. The detailed analysis results are shown in Table 1 and are explained in the following section.

Table 1: Overall Detection on Malicious Samples

	ILC detected		**no** ILC detected	
	Samples	Fraction	Samples	Fraction
PATTERN	6,658	91.5%	—	—
CRASH	20	0.3%	15	0.2%
PROCESS	83	1.1%	33	0.4%
DIALOG	0	0.0%	295	4.1%
NOTHING	20	0.3%	154	2.1%
Total	6,781	93.2%	497	6.8%

6.3 Discussion

Given the benign and malicious data sets as stated above we end up with a false positive rate of 0% and a false negative rate of 6.8%. However, we have seen a lot of samples which were broken and, though containing malicious content, were not able to produce malicious functionality when loaded into a PDF viewer. Furthermore, some samples only triggered their exploit when using a particular PDF application that was *not* contained in our application set. Finally, there exist some samples that perform malicious behavior that is not based on shellcode, but instead uses built-in features of the PDF viewer application. For instance, some samples redirect to malicious websites or exploit software bugs in third-party applications that can be started directly from within a PDF document. To fortify our results and argue why we assume an effective false negative rate that is much lower than the measured one, we analyzed the documents from the malicious data set in more detail.

6.3.1 Results without ILC Execution Detection

Our system failed to detect ILC execution for 497 documents from the malicious sample set. Despite this fact we assume that it works correctly and those *undetected* samples either target at an untested PDF viewer application or simply corrupted and do not function at all. Since we are not able to manually analyze about 500 samples, we have developed heuristics to prove (or at least find hints for) our assumption.

We first checked those 15 files with the *CRASH* label and found that all of them performed invalid memory accesses during parsing the document. Without further investigation we can not be sure if they are really malicious or simply corrupted. Nevertheless, since they crash before executing any shellcode they seem to labeled correctly by our system.

We then examined those 33 samples that created a new process (*PROCESS*) and we discovered that in all cases regular built-in features were used for that purpose. The started applications were the *Command Shell*, *Internet Explorer*, and, *Outlook Express*. In the latter two cases the used command line parameters [36] were specially crafted to enforce a parsing error and arbitrary code execution within those started applications.

After that we investigated the 295 cases that where labeled with *DIALOG*. Most of the dialogs contained error messages [36] of the parsing engine, which state that the PDF structure itself or some embedded JavaScript-code was invalid. We assume that either the corresponding PDF documents were corrupted or that we simply have not used the expected environment to trigger the malicious functionality. In addition to this, there are some PDF exploits that solely are based on social engineering, in which the user is tricked to respond in a particular way to the shown dialog. For example the warning message for starting a new process is obfuscated in a way such that the user will not notice that a new process will actually be spawning when he clicks the *OK* button [29, 30]. Overall, it is unclear whether these files were indeed malicious or not. However, it is probable that none of them executed any form of ILC during execution.

Finally, 154 samples did not perform any suspicious activity at all (*NOTHING*). Obviously, this does not mean necessarily that the samples are harmless. It just means that under the given environment they behaved benign. We manually checked a random set of 30 samples of this class and we found that they do not contain any working exploit at all. A reason for the AV scanners to mark them as malicious may have been that they contained malicious signatures as a pure coincidence or were the results of failed attempts to create working malicious PDF documents.

In summary, it is safe to assume that in all 497 cases really no ILC was executed in any of our used environments. So while our method failed to flag these samples as malicious, it succeeded in detecting the execution of ILC: since no ILC was executed no detection was triggered. In this sense all negatives seem to be true negatives, and so after careful consideration one could also claim a false negative rate of 0% for our approach and the examined sample set.

6.3.2 Results with ILC Execution Detection

For completeness, we also discuss the cases where our method detected ILC execution. In order to show that these cases are correct, we have to ensure that all dumped memory really consists of illegitimate code and that no prior ILC execution has been missed. Again, confirming this for each individual case is impossible due to time restrictions and, therefore, again we used heuristics to get trustful hints for the correctness. In the 6,658 cases that are labeled *PATTERN* we confirmed the presence of known shell code patterns in the dumped memory pages. Therefore, we can be sure that in fact ILC was executed.

We checked those 20 samples that crashed the PDF viewer after the ILC detection (*CRASH*). This is also an evident sign for (a partly failed) malicious activity. In such cases the exploit did not work well, either because it was badly programmed or because it did not discover the environment which was needed to work correctly. Even if the samples do not succeed to perform any reasonable malicious operation, we know that the observed code execution really is related to ILC and, accordingly, is no false positive.

Next we investigated those 83 documents that spawned new processes after the ILC execution (*PROCESS*). This can also be seen as a clear sign of malicious activity, if the started process is none of those mentioned in section 6.3.1, which could be started by legitimate built-in features of the PDF viewer. We have verified that none of the spawned processes belong to those exceptions, but all fell into one of the following three categories[36]: it was either tried to start an extracted or downloaded program (with malicious content), to open a created second PDF document (to hide the maliciousness of the initial document) or to gather essential information about the exploited system. Accordingly, we can be sure that all of these samples really have executed shellcode.

Finally, there were 20 remaining samples with ILC execution, for which all of our previously described heuristics failed. Hence, we were not able to tell anything about their maliciousness in an automated way and, therefore, we checked them manually. All of these files really executed ILC, from which some was simply not working correctly and other did not even consist of valid machine instructions at all. We can only guess the reasons for that: most probably, some of these samples just were written badly or got corrupted due to some transmission error. Others may find some unexpected environment and, accordingly, do not function properly. Anyway, we had manually assured ourselves that in all cases ILC was executed, no matter if the resulting operations were valid or not.

6.3.3 Detection Summary

Though we are not able to manually verify all the samples we have analyzed with our system, the results shown in the previous subsections lead to the conclusion that our approach works well. If we aggregate all our findings with illegitimate code execution, we get a minimum detection rate of 93.2% for our particular set. If we furthermore assume that there is a serious fraction of samples that does not contain working shellcode for any of our used environments, we can assume an error-corrected detection rate that is much higher in reality.

6.3.4 Detection Results of Other Analyzers

To evaluate the effectiveness of our solution, we compared our results against those from the popular application specific analyzers *Wepawet* [6], *pdf examiner* [34], and *ADSandbox* [7]. All of these analyzers combine static and dynamic approaches, i.e., they parse the PDF document structure, extract potential malicious pieces, and then analyze them by different means. Depending on the severity of the findings, each analyzed sample is labeled as either *benign*, *suspicious*, or *malicious*. Furthermore, additional comprehensive analysis data is generated, i.e., information about the embedded objects, like PE files, URLs, or known exploits.

Wepawet [6] combines machine learning techniques with emulation. It extracts specific features while emulating Javascript code and then compares them against a set of pre-

Table 2: Detection Results on Malicious and Benign Samples

	Malicious Sampleset				Benign Sampleset			
Analyzer	Malicious		Suspicious		Malicious		Suspicious	
Wepawet	4,737	(65.1%)	1,739	(23.9%)	0	(0.0%)	0	(0.0%)
pdf examiner	6,089	(83.7%)	1,108	(15.2%)	82	(1.1%)	246	(3.4%)
ADSandbox	2,360	(32.4%)	255	(3.5%)	0	(0.0%)	3	(0.1%)
CWXDetector	6,781	(93.2%)	0	(0.0%)	0	(0.0%)	0	(0.0%)

viously learned known benign profiles. It also uses a set of signatures to detect anomalies, which are *not* based on Javascript. *pdf examiner* [34] as well extracts all embedded objects and streams from the PDF document and decrypts them if necessary. It then uses signature scanning to detect known malicious patterns and *libemu* [3] to detect shellcode. From all the findings a score value is calculated, that decides about the ultimate outcome of the analysis. Besides this value, a sophisticated report is generated that highlights suspicious and malicious parts of the PDF document. In contrast to the two aforementioned analyzers, *ADSandbox* [7] solely aims at the detection of malicious Javascript within the PDF documents. For that purpose, all Javascript snippets are extracted and then executed in an isolated environment. Subsequently, heuristics are utilized to decide from the executed operations and the involved data about the maliciousness of the particular sample. ADSandbox can be used with several different configurations settings, but we have used the defaults for simplicity.

When comparing the results of these application specific analyzers to those created by *CWXDetector*, one has to take into account that our tool only triggers on the actual execution of ILC. Accordingly, it is only capable to label a sample as *benign* or *malicious*, but not as *suspicious*. Furthermore, all of the other analyzers work with heuristics and patterns and, therefore, are presumably not able to detect all kinds of unknown exploiting methods.

Table 2 summarizes all results for the detection of malicious samples. When only taking those samples into account that have been marked as *malicious*, our approach yields better results than those of the application specific analyzers. However, also when considering the suspicious samples as well, our results are comparable, i.e., 89.0% (Wepawet) and 98.9% (pdf examiner) vs. 93.2% (*CWXDetector*). Furthermore, we know that a significant part of those malicious samples which have been not detected by *CWXDetector* are corrupted and, hence, not executable at all. A signature scanning based approach is obviously able to detect malicious parts within those broken files, but our method obviously fails on them. ADSandbox does not deliver a very high detection rate on our malicious sample set since its main focus is to analyze JavaScript code only.

When it comes to the false positive rate, the comparison is rather simple (see Table 2). As described above our approach does not produce any false positive on the used sample set. Also the other three analyzers generate good results: 0 false positives for Wepawet as well as for ADSandBox. There is a trade-off in detection accuracy of pdf examiner, since this detector has the best detection rate but also produces the most false positives of around 4.5%, which still is an acceptably low number.

6.4 Additional Experiments

In order to emphasize the universality of our approach, we briefly present detection results from different applications. We used *CWXDetector* to analyze malicious *Flash* documents as well as malicious network packets that exploit vulnerabilities in the *Real VNC viewer* and the *VideoLan Client (VLC)* tools. A more detailed description of our findings is contained in the extended version of this paper [36].

6.4.1 Flash Documents

As additional example for shellcode containing documents we have analyzed two malicious *Flash* files. The first one was created with the help of the *Metasploit Framework*[21] and the other one was *JOB_DESCRIPTION.doc*, a sample that was found in the wild and taken from the *Contagio Dump Archive*[5]. Both samples exploit the *CVE-2011-0611* vulnerability of the *Flash Player* version 10.0.45 by executing a malicious *ActionScript* that results in arbitrary code execution. Since both *Flash* samples were embedded in *Word* documents, we has to use *Microsoft Word Professional 2010* to actually view them. Both samples were detected correctly by our tool and in both cases the memory pages containing the ILC were dumped.

6.4.2 VNC Client

The traditional way to execute shellcode on remote systems has been to embed it into network packets and exploit vulnerabilities in the parsing application. The increased awareness and improved security of contemporary network applications and operating systems has driven the attackers to shift to malicious documents. However, in order to illustrate the effectiveness of our generic approach we show that it is capable to detect malicious code execution also in this traditional context. Therefore, we used the *Metasploit Framework* to setup a network server that accepts connections from *VNC* clients. After executing the *RealVNC client version 3.3.7* in combination with our *CWXDetector*, we connected to that server and received a specially crafted network packet. This packet contained an exploit for the *CVE-2001-0167* vulnerability of the *RealVNC* application. *CWXDetector* detected the execution attempt of the first contained shellcode instructions and dumped the related memory to a file.

6.4.3 VideoLan Client

An additional analysis of a network application exploited by a malicious network packet was performed with help of the *CVE-2010-3275* vulnerability, which existed in the versions 1.1.4 up to 1.1.7 of the *VideoLan Client* (VLC). By accessing a specially crafted *.amv* file, VLC can be crashed by the usage of an invalid pointer and arbitrary code can be executed. With the help of *Metasploit* we again set up a server that generated such malicious data and offered it

over the network for download. Unsurprisingly, our detection mechanism triggered again and extracted the malicious instructions once the embedded shellcode was about to be executed.

7. EXTRACTION EVALUATION

In this section we try to measure the *quality* of the extracted ILC. For that purpose we determined the percentage of contained valid x86 instructions (*code ratio*) and the amount of data in terms of embedded strings (*data ratio*). Since shellcode often uses encryption and code obfuscation to avoid detection, we expected only poor *quality* when investigating the initially created dump files. To encounter this problem, we applied the MVD feature described in Section 4.5. To reduce the amount of information to be examined, we only used a subset of the malicious PDF documents and only one particular viewer. More specifically, we chose *Adobe Acrobat Reader 9.00* and those 4,869 samples for which ILC execution was detected in that viewer in the first experiment. In particular, we performed the following steps to measure the quality of the extracted ILC:

- we analyzed 4,869 PDF samples in *Acrobat Reader 9.00* with enabled MVD,
- all consecutive memory pages were concatenated to *code regions*, resulting in either one or two versions each:
  - one *initial region*, if only one dump version of each contained memory page exists,
  - and additionally one *final region*, if more than one dump version exists for at least one contained page
- the code ratio of each region was determined by using *IDA Pro*, and
- all valid strings from each region were extracted and counted.

We then selected those code regions for which an initial *and* a final version existed, i.e., those which contain self-modifying code. We found 2,534 regions of this kind. Figure 2 illustrates the code ratio for the initial and final regions, respectively. One can easily see that the percentage of valid instructions increases dramatically when applying the MVD feature. Figure 3 shows the improvement of the data ratio – in terms of valid strings – when the shellcode is deobfuscated in an automated way. When comparing with the code ratio, the improvement is only marginal. Nevertheless, in sum we extracted 7,807 strings and 1,866 URLs from the initial regions, and 8,676 strings and 2,280 valid URLs from the final ones.

8. LIMITATIONS

The approach described in this paper is solely based on dynamic analysis of the examined malware samples. Therefore, it suffers from all the drawbacks and limitations of dynamic analysis in general. Since during each code execution only one particular control path is taken, the gained results always may be incomplete. If a required environment condition is not met and, therefore, a certain malicious functionality is not triggered during execution, dynamic analysis is unable to reveal any information about it. Accordingly, our system is incapable to detect embedded malicious code in general, but only detects it when it gets executed. Obviously, our system is not meant to protect end consumer

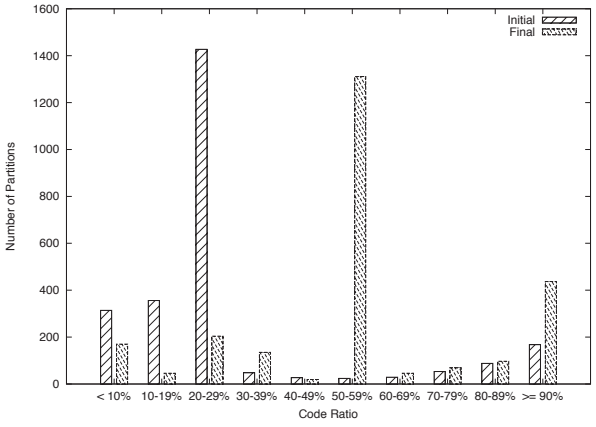


Figure 2: Code Ratio Distribution

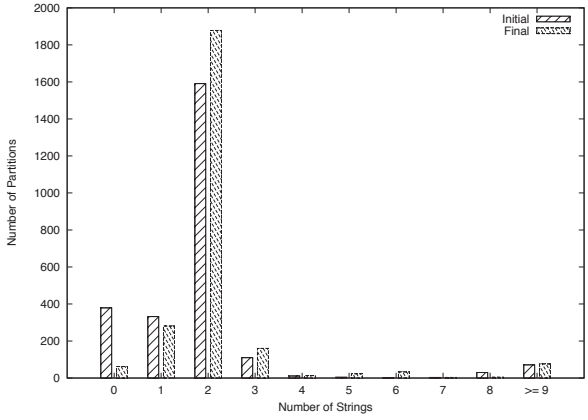


Figure 3: Data Ratio Distribution

hosts, but its sole purpose is to support malware analysis on dedicated analysis systems.

Furthermore, the existence of malicious computation does not always imply the existence of illegitimate code. Therefore — and similar to DEP — our approach has problems with novel exploitation techniques like *return oriented programming* (ROP) [24] or *JIT-spraying* [4, 25]. However, we share those limitation with systems that are similar to ours and, furthermore, advanced attacks usually consist of multiple stages of which only the first uses ROP/JIT-spraying to set up a later stage comprising regular illegitimate code which then can be detected and extracted using our method. Nevertheless, despite its difficulty it is possible to create full ROP-shellcode that are undetectable with out approach.

9. CONCLUSIONS

In this paper, we presented a generic and automatic method to detect and extract illegitimate code during an attack. We introduced *CWXDetector*, an implementation of our approach for the 32 bit Windows XP version, and evaluated it by analyzing a large corpus of malicious PDF documents. Our system turns out to be very effective in supporting malware analysis, since the detection rates improve state-of-the-art tools and it directly supports the analyst by extracting

a small set of memory pages for manual inspection. We also evaluated *CWXDetector* with different kinds of other file formats and even complex attack vectors like a Flash file embedded in a Word document can be analyzed successfully.

From an analyst point of view, especially the contained URLs and server host addresses that point to additional malware sites are valuable resources. Furthermore, the insights gained by a post processing analysis may assist in developing new protection techniques and creating signatures for zero-days until patches are available. We have also shown how the quality of the extracted ILC can be increased dramatically by applying multi-version dumping to automatically deobfuscate it.

Acknowledgments.

We would like to thank Tilo Müller for reading earlier versions of this document and making helpful suggestions for improvements. Additional thanks go to Andreas Dewald, Marco Cova, and Tyler McLellan for their great support while using their analysis tools. This work has been supported by the German Federal Ministry of Education and Research (BMBF grant 01BY1205A – JSAgents).

10. REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2005.

[2] P. Akritidis, E. P. Markatos, M. Polychronakis, and K. Anagnostakis. Stride: Polymorphic sled detection through instruction sequence analysis. In *20th IFIP International Information Security Conference*, 2005.

[3] P. Baecher and M. Koetter. libemu - x86 shellcode detection and emulation, 2007. `http://libemu.carnivore.it/`.

[4] Dionysus Blazakis. Interpreter exploitation. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2010.

[5] contagio Website. Malware Sample Dump For CVE-2011-0611 Flash Player Zero day. `http://contagiodump.blogspot.com/2011/04/apr-8-cve-2011-0611-flash-player-zero.html`.

[6] Marco Cova, Christopher Kruegel, and Giovanni Vigna. Detection and Analysis of Drive-by-download Attacks and Malicious JavaScript Code. In *World Wide Web Conference (WWW)*, 2010.

[7] Andreas Dewald, Thorsten Holz, and Felix C. Freiling. ADSandbox: Sandboxing JavaScript to fight malicious websites. In *ACM Symposium on Applied Computing (SAC)*, 2010.

[8] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *ACM Conference on Computer and Communications Security (CCS)*, 2008.

[9] Adobe Systems Incorporated. Document management, portable document format, part 1: Pdf 1.7, 2008.

[10] Intel Corporation. Intel: 64 and IA-32 Architectures Software Developer's Manual. Specification, Intel, 2007. `http://www.intel.com/products/processor/manuals/index.htm`.

[11] Christopher Jordan. Writing detection signatures. *USENIX ;login:*, 30(6):55–61, 2005.

[12] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. Renovo: a hidden code extractor for packed executables. In *ACM Workshop on Recurring Malcode (WORM)*, 2007.

[13] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure execution via program shepherding. In *USENIX Security Symposium*, 2002.

[14] Lionel Litty, H. Andrés Lagar-Cavilla, and David Lie. Hypervisor support for identifying covertly executing binaries. In *USENIX Security Symposium*, 2008.

[15] Lorenzo Martignoni, Mihai Christodorescu, and Somesh Jha. Omniunpack: Fast, generic, and safe unpacking of malware. In *Annual Computer Security Applications Conference (ACSAC)*, 2007.

[16] Microsoft. Enhanced mitigation experience toolkit (EMET). `http://support.microsoft.com/kb/2458544/de`.

[17] MSDN. A detailed description of the data execution prevention (DEP) feature. `http://support.microsoft.com/kb/875352/en-us`.

[18] Udo Payer, Peter Teufl, and Mario Lamberger. Hybrid engine for polymorphic shellcode detection. In *Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, 2005.

[19] Michalis Polychronakis, Kostas G. Anagnostakis, and Evangelos P. Markatos. Network-level polymorphic shellcode detection using emulation. In *Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, 2006.

[20] Sebastian Porst. Dumping shellcode with Pin. `http://blog.zynamics.com/2010/07/28/dumping-shellcode-with-pin/`.

[21] Rapid7. The metasploit framework. `http://metasploit.com/`.

[22] Karthik Selvaraj and Nino Fred Gutierrez. The rise of PDF malware. `http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/the_rise_of_pdf_malware.pdf`, 2010.

[23] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *ACM SIGOPS Symposium on OS Principles (SOSP)*, 2007.

[24] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[25] Alexey Sintsov. Writing JIT-spray shellcode for fun and profit. `http://dsecrg.com/pages/pub/show.php?id=22`.

[26] Hispasec Sistemas. Virus total. `http://www.virustotal.com/`.

[27] Kevin Z. Snow, Srinivas Krishnan, Fabian Monrose, and Niels Provos. SHELLOS: enabling fast detection and forensic analysis of code injection attacks. In *USENIX Security Symposium*, 2011.

[28] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, Newso James, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *International Conference on Information Systems Security (ICISS)*, 2008.

[29] Didier Stevens. `http://blog.didierstevens.com/2010/03/29/escape-from-pdf/`, 2010.

[30] Didier Stevens. `http://blog.didierstevens.com/2010/03/31/escape-from-foxit-reader/`, 2010.

[31] Joe Stewart. OllyBone: Semi-Automatic Unpacking on IA-32. *Defcon 14*, 2006.

[32] PaX Team. Documentation for the PaX project - overall description. `http://pax.grsecurity.net/docs/pax.txt`, 2008.

[33] The Pax team. PaX address space layout randomization (ASLR). `http://pax.grsecurity.net/docs/aslr.txt`, 2003.

[34] Malware Tracker. pdf examiner. `http://www.malwaretracker.com/pdf.php`.

[35] Carsten Willems. Windows memory management internals (not only) for malware analysis. Technical report, University of Mannheim, 2011.

[36] Carsten Willems and Felix C. Freiling. Using memory management to detect and extract illegitimate code for malware analysis. Technical report, University of Erlangen, 2012.

Down to the Bare Metal: Using Processor Features for Binary Analysis

Carsten Willems, Ralf Hund, Andreas Fobian, Dennis Felsch, and Thorsten Holz
Ruhr-University Bochum, Horst Görtz Institute for IT-Security (HGI)
{firstname.lastname}@rub.de

Amit Vasudevan
Carnegie Mellon University
amitvasudevan@cmu.edu

ABSTRACT

A detailed understanding of the behavior of exploits and malicious software is necessary to obtain a comprehensive overview of vulnerabilities in operating systems or client applications, and to develop protection techniques and tools. To this end, a lot of research has been done in the last few years on binary analysis techniques to efficiently and precisely analyze code. Most of the common analysis frameworks are based on software emulators since such tools offer a fine-grained control over the execution of a given program. Naturally, this leads to an arms race where the attackers are constantly searching for new methods to detect such analysis frameworks in order to successfully evade analysis.

In this paper, we focus on two aspects. As a first contribution, we introduce several novel mechanisms by which an attacker can *delude* an emulator. In contrast to existing detection approaches that perform a dedicated test on the environment and combine the test with an *explicit* conditional branch, our detection mechanisms introduce code sequences that have an *implicitly* different behavior on a native machine when compared to an emulator. Such differences in behavior are caused by the side-effects of the particular operations and imperfections in the emulation process that cannot be mitigated easily. Motivated by these findings, we introduce a novel approach to generate execution traces. We propose to utilize the processor itself to generate such traces. Mores precisely, we propose to use a hardware feature called *branch tracing* available on commodity x86 processors in which the log of all branches taken during code execution is generated directly by the processor. Effectively, the logging is thus performed at the lowest level possible. We evaluate the practical viability of this approach.

1. INTRODUCTION

During a typical attack against a computer system, an attacker first exploits some kind of (software) vulnerability to gain access to the system. Once she has control over the compromised machine, the next step is to install some kind of malicious software (abbr. *malware*) that, for example, steals sensitive information or hides the presence of the attacker on the system. Both software vulnerabilities and malware are thus closely linked and we need to have a precise understanding of their semantics to combat this threat. Most importantly, detailed analysis reports about the tools used by an attacker are required to fix vulnerabilities in operating systems or applications, and to develop new protection techniques and tools.

Nowadays, antivirus companies analyze tens of thousands of malware samples on a daily basis [41] with new exploits being released frequently. Thus, there is a clear need for automated approaches to analyze these threats. As a result, a lot of research has been done on efficiently and precisely analyzing malicious code both in academia and industry (e.g., [2, 3, 6, 7, 10, 11, 22, 27, 40, 44]) and many tools and techniques for automated analysis are currently available. The analysis of malicious and vulnerable code can be implemented in several ways and on several semantic levels. Broadly speaking, the methods can be divided into *static* and *dynamic* approaches, each having their own (dis-) advantages. For example, static analysis is often complicated with code obfuscation and encryption [24, 31, 39], whereas dynamic analysis is typically only capable of efficiently examining a limited number of execution paths [27].

A very detailed behavioral view of code can be obtained by examining every single instruction, but this approach produces a huge amount of data, which has to be mined for valuable information. On the contrary, monitoring only the system calls performed by the program achieves a smaller analysis data set, but results in a high abstraction level and can be evaded in many different ways [13], leading to incomplete analysis results. From a performance perspective, single stepping a program to perform an instruction-level analysis is much slower than intercepting only system calls.

Furthermore, malware analysis can be implemented on a native machine (also called *bare metal approach*) or in an emulated/virtualized one. When using a native machine, we are faced with several problems. Most importantly, the analysis system may get infected by malicious code and it has to be reverted back to a clean state after the analysis process has finished. Furthermore, most machines only offer rudimentary monitoring facilities and additional mechanisms have to be implemented first. Sophisticated approaches use techniques like dynamic translation

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

(e.g., *Cobra* [44]) or hardware virtualization extensions (e.g., *Ether* [10]) to achieve such monitoring.

In contrast, emulators pose a powerful trade-off between performance and convenience with respect to native machines, but they lack transparency and correctness. Many malware authors have come up with a variety of detection mechanisms that uncover the presence of such artificial environments [13,14,30,32,35] and several systematic studies on detecting virtual machines or CPU/system emulators have been performed [25,26,32]. Once the malware has detected the presence of the analysis environment, it can behave differently leading to incorrect analysis reports. Apart from these explicit detection techniques employed by malware, there are also different CPU instructions [35] and real-life conditions (e.g., timing aspects or specific artifacts like the username of the analysis machine) under which a binary might behave differently when executed inside an emulated environment as opposed to a native system.

In this paper, we continue this line of work and present mechanisms an attacker can use to implement code that behaves differently in the presence of an analysis environment. Our mechanisms are novel as they do not perform any *explicit* test on the analysis environment. Instead, we use instruction sequences that have different semantics on a real machine when compared to an emulated one. More precisely, such instruction sequences have an *implicitly* different behavior on a native machine with respect to an emulator due to the side-effects of particular operations and imperfections in the emulation process that cannot be mitigated easily (e.g., self-modifying code or caching effects). Effectively, our techniques delude the emulator and thus we call this approach a *delusion attack*.

We use delusion attacks as our motivation and propose to utilize hardware features of commodity x86 processors to overcome the (accidental or intended) incorrectness of dynamic analysis in an emulated environment. More precisely, we introduce a promising approach to analyze the behavior of binary programs by using a processor feature called *Branch Tracing* (BT). With this hardware primitive (available on both Intel and AMD CPUs [17]), the processor *itself* keeps track of all branches taken during code execution. The logging is thus performed at the lowest level possible, making our approach robust to attacks. Our performance overhead is also significantly lower in contrast to other approaches that use hardware features such as single stepping [10].

To demonstrate the effectiveness and applicability of our approach, we show how our method is sufficient to analyze malicious PDF documents. In an empirical evaluation, we demonstrate that the branch tracing results can be used to automatically cluster similar vulnerabilities which are exploited within the analyzed documents: a set of 4,869 PDF documents can be clustered into eight different root causes based on the analysis results of our tool. Most notably, our framework can also deal with advanced exploits that use concepts like structured exception handler (SEH) for control flow diversion and even return-oriented programming [37].

Related Work.

As discussed previously, there is a large body of published work on malware analysis and detection of different execution environments. Complementary to our work are recent approaches that compare the behavior of a sample in different (analysis) environments [2, 20, 21]. Such techniques could also be used to detect our delusion attacks, but they incur huge runtime overheads as a single sample has to be executed in at least two analysis environments. Vasudevan et al. introduce a way to use branch tracing on AMD CPUs to record host execution trace to an external, trusted system [43]. In contrast, we also show how BT can be used on Intel CPUs and perform several empirical experiments to demonstrate the practical usefulness of this approach.

Contributions.

The main contributions of this paper are as follows:

- We introduce several delusion attacks for software emulators. These instruction sequences behave differently when executed on a native machine as opposed to an emulator. Delusion attacks work by exploiting some implicit imperfections in the emulation process.
- Motivated by delusion attacks, we introduce an approach to perform behavior analysis that takes advantage of the *branch tracing* feature of commodity x86 CPUs. Our approach performs the logging of the actual behavior on the lowest level possible since we directly instrument the CPU to generate traces.
- We have implemented a fully-working prototype of our approach and show in an empirical evaluation the usefulness of our approach by performing a crash analysis of malicious PDF documents.

Technical Report.

Due to space limitations, we have published an extended version of this paper as an accompanying technical report [47]. In that paper, we introduce more delusion attacks, describe the processor BT facilities in detail, and explain the payload of a practical delusion attack.

2. SOFTWARE EMULATORS

A lot of research has been focused on malware analysis in the past. Accordingly, many different techniques have evolved in this field, e.g., the application of debuggers and recently also hypervisors or binary instrumentation. Nevertheless, software emulation-based solutions are oftentimes more appealing for malware analysis since they provide full control over the emulated system: the analyzer can intervene at any point in the execution of the analyzed code. There are also no restrictions on analyzing privileged code within the guest. Furthermore, emulators provide isolation between the analyzer and the malware.

In this section we thus will review existing emulation techniques that are used by malware analysis frameworks. This serves as a discussion of related work and motivates our delusion attacks that we introduce in Section 3. A more detailed description of other analysis techniques can be found in the extended version of this paper [47]. There, we shed light on the advantages and disadvantages of each approach and provide some examples of tools and methods that have been proposed in the literature.

BOCHS [23] is a PC emulator that emulates an x86/x64 processor with a set of common attached devices (graphics card, network card, etc.). It is an emulator in the classical sense in that the emulated code is fetched, decoded, and emulated instruction-wise — implemented in one large loop in the *BOCHS* code. This enables a precise emulation of the guest system, but has the drawback that execution is

typically slow in comparison to a real system. *BOCHS* is often used for malware analysis together with the disassembler IDA Pro [36], which ships with built-in support for that purpose. This combination can efficiently be used to (partially) execute malware within the emulator to analyze it.

Similar to Bochs, *QEMU* [4] is a generic full system emulator. However, by using an intermediate language and a technique called *dynamic translation*, it achieves support for a variety of host and target platforms along with good performance. *QEMU* is thus considerably faster than other emulators. The dynamic translation engine works as follows: whenever new code is executed in the emulator, *QEMU* translates the corresponding block of instructions (i.e., the instructions until the next branch) into an internal intermediate language. From this representation, the code is optimized to reduce unnecessary overhead (e.g., setting certain flags that are not further evaluated anyway) and then translated into the final, architecture dependent target code. The resulting block of code is called *translation block* (TB). TBs are cached so that the translation process is ideally only executed once. Guest code memory accesses are translated into safe memory accesses in the target code such that they cannot escape from the isolated, emulated memory space. Self-modifying code is detected with the help of a page fault exception handler. To this end, executable pages of the emulated guest are marked as non-writable. Whenever translated code attempts to overwrite code that corresponds to a TB, an exception within *QEMU* happens and the emulator invalidates all affected TBs.

QEMU forms the basis of several malware analysis platforms, such as the dynamic analyzer of *BitBlaze* [40] (called *TEMU*) and *Anubis* [3]. Amongst other things, *TEMU* and *Anubis* extend *QEMU* by providing *taint propagation tracking* [28], a technique that allows to backtrack which input values influenced the value of a certain register, memory location, or similar storage units. In order to do so, every translated write operation has to be instrumented and dependencies between memory values have to be saved in a dedicated internal memory region. Taint propagation tracking thus comes with a significant performance penalty. These frameworks also introduce OS awareness through *virtual machine introspection* [15]. This provides access to runtime information such as running processes or loaded drivers, and also allows to hook specific events in the emulated system such as certain API calls. Since both systems rely on the emulation code of *QEMU*, they are both prone to design- or implementation-related flaws of *QEMU*.

3. IMPLICIT METHODS TO DELUDE SOFTWARE EMULATORS

Obviously, attackers have an incentive to evade automated malware analysis frameworks. Thus, there is an arms race in this area where the attackers are constantly searching for new methods and techniques to detect such analysis frameworks. To this end, different techniques to detect emulators were introduced in the last few years [13, 14, 30, 32, 35] and several systematic studies on detection approaches were performed [25, 26, 32]. As a result, there is a large body of work on detection approaches and attackers have plenty of ways to detect the presence of a virtual environment. In the following, we introduce another approach in which code implicitly behaves differently on a native machine compared to an emulated one, a technique we call *delusion attack*.

3.1 Motivation

To motivate the benefit of utilizing hardware features for dynamic program analysis, we propose a new class of emulator detection techniques. Current methods consist of two different steps: first, the existence of a non-native system environment is probed and then, depending on the outcome of the test, different actions are performed. These detection attempts are easy to spot and mitigate during (manual or automated) examination of the performed operations [44]. In contrast, our methods have no explicit check and do not contain a conditional branch that takes one control path on a native machine and another on an emulated one. Even powerful analysis techniques like multi-path execution [27] cannot analyze this kind of code sequences since the emulator *itself* does not execute the correct code. We call this *delusion attacks* to emphasize the fact that such code sequences effectively delude emulators in the way code is interpreted.

All our examples that we present follow the same methodology: a sequence of instructions is executed and as an implicit effect, either a malicious or a benign function is called. In a real attack, the malicious instructions are executed directly inline instead of calling a separate function. For the sake of simplicity we use two dedicated subfunctions in our examples: *MALICIOUSCODE* (that should executed on a native system) and *BENIGNCODE* (to be executed in an emulator). The examples were implemented and validated against real hardware as well as the targeted emulators.

There are hundreds of different processor types (including the different steppings of CPUs) available that vary in small implementation details. Due to this fact, we were not able to test all of them in an empirical evaluation. Hence, we focus our analysis on common Intel and AMD CPUs and, as a result, we cannot claim that these techniques are universally usable and provide a way for a guaranteed emulator delusion. The goal of this work is to show that there are still and – most probably – will always be methods to exploit behavior differences of emulators in order to force different execution for the same piece of code. As a result, we argue that it is not safe to trust analysis results that are gathered with the help of emulators since a program might behave differently on a native machine.

Note that traditional detection techniques could be modified in such a way that conditional code branches are transformed into *branchless code* as well. Therefore, the boundary between those and our new delusion methods is blurred to a certain extend. Nevertheless, we are confident that our techniques are significantly harder to detect and mitigate compared to previous approaches.

3.2 Basic Principle: Self-Modifying Code and Atomicity

Several of our attacks are based on *self-modifying code* (SMC). Correct handling of SMC is a non-trivial and complicated task when not done by the CPU itself, but by an emulator. Thus we expect that an attacker can use SMC to detect the presence of an emulator.

On a native system, the modification of data within a code segment has to trigger different actions. Most importantly, the old version of the modified code has to be flushed

from the *instruction prefetch queue* and from the *instruction caches*. Depending on the underlying cache organization and the number of processors available within the system, also the other CPUs have to be informed and take care of this problem accordingly.

Contemporary systems contain sophisticated measures to detect SMC correctly and there have been many flaws in the past that required the developer to perform specific actions in order to realize properly working code. For example SMC typically only operated correctly if (after modifying the memory) either a jump operation to that modified code or a memory-serializing operation (e.g., `cpuid` instruction) had been performed. Additional problems occurred with instructions that had already been loaded into the instruction prefetch queue: since only the linear address of a modified memory location was checked, it was possible to use two different linear addresses for code and data access, which both are associated with the same physical memory.

Modern CPUs can handle these older problems with SMC correctly. In an emulator, however, the CPU facilities for SMC detection obviously cannot be utilized. Hence, they have to be emulated and implemented in software as well. One way is to check every memory write operation against a list of addresses that possibly contain instructions or vice versa when an instruction is about to be executed. Apparently, this implies a huge overhead of execution performance and space required for managing the related data structures. Thus, most emulators use page fault handling for SMC detection. To this end, all executable memory pages are marked as *read-only*. If the emulated code performs a write attempt to such a memory area, the page fault handler is triggered. It performs the following actions if the target memory *should* be writable, i.e., if it was marked read-only by the emulator and not by the application itself:

1. all other threads are suspended,
2. the memory protection is modified to *writable*,
3. the faulting write instruction is executed again,
4. the memory protection is changed to *read-only*, and
5. all other threads are resumed.

This approach is problematic due to the fact that emulated instructions are normally not executed *atomically*, but they are translated into several sub-instructions. Therefore, the faulting write operations may already be partially executed and then have to be re-executed after the memory is made writable. This behavior can be exploited for delusion purposes as shown in the following subsection.

3.3 REP MOVS Instruction

The first delusion method uses the `rep movs` instruction, which copies a number of bytes, words, or double words within an implicit loop. The source memory location is specified by the `esi` register, the destination location by `edi`, and the amount of copy iterations by the value of `ecx`. This value is decremented with each copy iteration and used as a stop condition once it reaches zero. Accordingly, the value of `ecx` equals to 0 when the complete loop is finished.

To delude an emulator, the copy destination can be set to the memory address of the `rep movs` instruction itself. As an effect, this instruction is overwritten by the first copy iteration. On a real machine, the copy loop is performed atomically, so this instruction overwriting has no actual effect on the loop execution. After the copy operation is completed, `ecx` is zero and the `rep movs` instruction, as well as

```
        lea eax, BENIGNCODE
        lea ebx, MALICIOUSCODE
        lea esi, NEW
        lea edi, OLD
        mov ecx, 2
OLD+0x0: rep movsd
OLD+0x2: nop
OLD+0x3: nop
OLD+0x4: call eax   // => BENIGNCODE
OLD+0x6: nop
OLD+0x7: nop
        ret

NEW+0x0: nop
NEW+0x1: nop
NEW+0x2: nop
NEW+0x3: nop
NEW+0x4: call ebx   // => MALICIOUSCODE
NEW+0x6: nop
NEW+0x7: nop
```

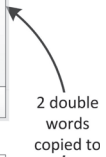


Figure 1: Delusion attack with a `rep movs` instruction.

the consecutive ones, are overwritten. In an emulator, however, the situation is different: due to the detection mechanism already the first loop iteration triggers the page fault handler when trying to write to memory that contains code. The emulator makes the destination memory writable and re-executes the memory write operation. Afterwards, the instruction is re-read from memory, in order to not miss any SMC. Since now the re-fetched instruction is no longer `rep movs`, a different behavior arises when compared to a real machine. For example, if the instruction is overwritten with `nop` operations, only one single copy loop iteration is performed: only the `rep movs` instruction is overwritten and the following instructions remain untouched. Furthermore, the `ecx` register is only decremented by one.

This different behavior can be exploited by the delusion code shown in Figure 1. Note that the `movsd` instruction copies one double word per iteration. On a real machine, two copy iterations are performed and, therefore, two double words are copied from *NEW* to *OLD*. After finishing, the memory at *OLD +0x4* contains the call to the *MALICIOUSCODE*. Accordingly, the malicious code is executed. Hence, on an emulated machine, the copy operation stops after overwriting the `rep movs` and the call to *BENIGNCODE* is *not* modified and, therefore, the benign code will be executed. *QEMU* and *BOCHS* can be successfully deluded with this technique.

We would like to stress that the deviating behavior can be fixed in the emulators. However, this would require special handling for a variety of instructions that can be used in conjunction with `rep`. This not only takes considerable effort to implement, but would reduce the performance of string copy operations in general. As an implementation detail note that not all CPU types behave similar when executing the code shown above. We found that some versions of the latest *Intel i7* CPUs *react* on the modification of the `rep movs` instruction and terminate the loop prematurely. In contrast, most other CPUs interpreted this code sequence in the way discussed above.

3.4 INVD Instruction

Besides SMC, there are other aspects of a system that are hard (if not impossible) to deal with when building an

```
  lea eax, BENIGNCODE
  lea ebx, MALICIOUSCODE
  lea esi, A
  inc esi
  wbinvd
  mov byte ptr [esi], 0xD0
  invd
A:
  call ebx       // FF D3 = call ebx
                 // FF D0 = call eax
```

Figure 2: Delusion with the help of the `invd` instruction.

emulator. One example are the many different kinds of caches available on a contemporary computer system. Some of these caches only contain data, others only instructions, and there are also combined caches that store both data and instructions. Furthermore, some caches are integrated into the CPU itself and others are placed outside (i.e., somewhere between the processor and the main memory).

Emulators cannot use these hardware facilities explicitly, since they need total control over the accessed data and the executed instructions. More precisely, emulators implicitly share the cache with the host operating system, since they also use the RAM for storing data and instructions. Nevertheless, *inside* the emulated system there is no explicit cache support and all cache-related instructions have no effect when being executed. Disabling all cache-related functionalities inside the emulated machine is the only reasonable way, since the simulation of caching facilities would degrade the performance of memory accesses even more and that would be counterproductive to the reason for using caches at all.

This missing ability to emulate cache is exploited by our third delusion technique. It works by utilizing *write-back* cache, which has no effect on an emulated machine. First of all, the instructions residing in a cached memory location are modified. On a real machine, the modification only affects the cache and the propagation to RAM is delayed for a while. On an emulated machine – and on each machine without caching – the modification is written directly to RAM. Now, immediately after modifying the memory, the cache is invalidated. On a real machine, this undoes the previous modification, while on an emulated one it (again) has no effect. Finally, the instructions within that memory buffer are executed and a different behavior between native and emulated machines is achieved. The actual code for this method is shown in Figure 2. For sake of simplicity, the instructions for enabling the caching for memory region *A* are left out. The code starts with writing back all potentially pending cache modifications to the RAM via the `wbinvd` instruction. Then the `call ebx` instruction is modified to `call eax`, which changes the call target from *MALICIOUSCODE* to *BENIGNCODE*. Afterwards, the instruction `invd` undoes this modification on a real machine, but not within an emulator. Finally, the call is executed to the resulting call target.

Although it is easy to detect such a scenario since `wbinvd` is an uncommon instruction, it takes vast effort to perfectly emulate the effects of `wbinvd`. This would require to provide a write history buffer that holds the recently written values to roll back the invalidation, which results in a significant performance degradation of the entire emulator. The instruction also needs elevated privileges (ring-0). However, this poses no problem since a full-system emulator can also be used to analyze privileged code. We verified that this kind of delusion attack works against both *BOCHS* and *QEMU*.

3.5 LEAVE Instruction

Our third method requires *virtual memory* and utilizes the x86/64 machine instruction `leave`, which behaves like the two instructions `mov esp, ebp` and `pop ebp` combined into a single operation. On a real machine, the `leave` instruction is always executed atomically. However, within an emulator it is possible to force a partial execution only. If for instance the `ebp` register initially is set to an inaccessible virtual memory address, this address will be correctly copied into `esp`, but the `pop` operation will trigger a page fault. If the emulator does not take special care of this situation, the `esp` value contains the overwritten value from `ebp` when entering the invoked exception handler. Obviously, on a real machine `esp` has not changed at all, since the `leave` operation could not be executed completely. A detailed description of this attack and appropriate example code to apply it can be found the technical report [47].

4. BINARY ANALYSIS WITH BRANCH TRACING

Motivated by the examples of detecting emulated environments, we now introduce an approach to observe the actual operations performed by a CPU. We then do not need to take into account the effects of SMC, caching effects, or other kinds of delusion and/or detection attacks. Effectively, this is the lowest level an analysis framework can be based on since we directly observe the behavior of the code when running on a CPU, i.e., we obtain a precise trace of the runtime behavior of the code.

As discussed before, the traditional way to trace a binary program operates on the instruction level. This can be achieved by either utilizing specific hardware features (e.g., single-stepping the CPU or virtualization features [10,34]) or by emulation [3,40]. For the single-stepping case, some specific debug control registers are set such that the CPU stops execution after each performed instruction and invokes an exception handler. This handler then can be used to examine or modify the processor registers or the memory, e.g., the heap or stack memory. Before returning control to the interrupted piece of code, the tracing can be re-enabled, since it is normally deactivated automatically after each handler invocation. Virtualization features of modern CPUs can be used to also generate single stepping traces, but this approach has a severe performance penalty in practice.

A more coarse-grained tracing granularity has been employed in the last years: when tracing on the *function-/system-call* level, execution is not stopped after each single operation, but only when specific functions are called. Often the set of monitored function calls is restricted to a subset of critical system calls. On interception, several actions can be taken. Typically, the call parameters are first examined and optionally modified. Then, the originally called function is executed or simulated. Finally, the result values are examined and/or modified appropriately. There exist several techniques for *function level* tracing. While native systems mostly apply API hooking [3, 40, 46], the usage of virtual machines empowers the analyst to perform *virtual machine introspection* (VMI) [3, 10, 15].

The granularity of tracing on the *branch* level is located between those of instructions and function calls. The interception happens on each taken branch, i.e., on each conditional and unconditional jump, call, interrupt, and exception. In the preceding the term *tracing* was used to describe a technique in which execution of a process is actually *stopped* after each instruction, branch, or, function call. However, with *BT logging* the execution is actually *not* interrupted, but only log information is stored at each interception point which results in a significantly smaller overhead. In practice, the performance overhead of BT logging is smaller than interrupting the actual execution (either via single-stepping the CPU or using virtualization features). Branch tracing provides a rather coarse overview of the behavior exposed by a given binary; the data which is collectable by BT logging is less comprehensive: the trace only contains the addresses of the source and target instructions of the branches. Nevertheless, even by viewing only addresses, it is possible to completely reconstruct the execution/decision path that was taken during execution as we will show in Section 5.

To implement a tracing framework, we take advantage of the *branch tracing* (BT) facilities available on x86/64 architectures from Intel and AMD. Though the implementation details for both platforms differ, the general approach is the same. Since only the Intel specifications contains detailed information on this topic [17], we mostly refer to the names and mechanism descriptions in that specification. In addition, we also learned how to use BT on the AMD platform through reverse engineering and empirical experiments.

We note that Intel and AMD both publish public documentation regarding light-weight processor performance monitoring mechanisms [1, 18]. While these mechanisms allow tracing of retired branch instructions, they are severely restricted in the information that can be captured. For example, Intel CPUs only log the last 4 to 16 branches. While AMD CPUs do not place any restrictions in terms of the number of branch instructions that can be profiled, they can only capture branches in user-mode (ring-3) and cannot capture far jumps, returns, sysenter/sysexit, exceptions, SMM mode etc [1].

A detailed description of the BT facilities of Intel and AMD platforms is given in the extended version of this paper [47]. There, we describe all the involved CPU registers and data structures and modes of operation.

5. APPLICATION OF BRANCH TRACING

In this section we describe several experiments that demonstrate the effectiveness of BT over traditional analysis approaches and show its wide applicability. We first show that the addresses contained in the BTMs are sufficient to reconstruct precise code paths taken during execution in the context of a practical and challenging application called *binning*. Here, we are interested in automatically grouping crash reports resulting from malware exploiting a vulnerability into different representative classes. After that we present how BTs can be enriched with additional information to obtain a deeper insight into executing code and demonstrate how complex return-oriented programming attacks [37] can be detected and analyzed with this approach. Finally, we describe a practical delusion attack that underlines the necessity of a robust tracing facility such as BT since traditional analysis approaches are unable to produce correct results.

5.1 Experiment 1: Binning of Malicious PDF Documents

One powerful application of BT is the grouping of crash reports gained by *fuzzing* [16, 29]. This kind of automated vulnerability analysis very often produces a large number of application crashes that are ultimately caused by the same software vulnerability. Since the post-verification of each single crash is very time consuming, an analyst wants to reduce the amount of reports to be examined as good as possible. For that purpose a technique called *binning* is used, in which the crash reports are automatically grouped into different classes, each one consisting of crashes that are a result of the same root cause. This saves a lot of time, since an analyst only has to manually analyze one instance of each resulting bin. One efficient way to realize binning is to compare the execution paths that have led to the crashes. Obviously, the specific part of a BT log that was protocolled just before an error has occurred contains all the necessary information for reconstructing the control path leading to the fault. The same technique can also be used to group a set of (possibly unknown) exploits by the root cause vulnerability they exploit.

Trace Generation.

For evaluating this method, we have extended a tool called *CWXDetector* [45] that is capable of detecting exploitation attempts and extracting shellcode used during the exploit. The tool uses a detection approach that is generic in the sense that it captures the fact that unauthorized code is being executed and is thus capable of detecting shellcode that is embedded and invoked from arbitrary types of data or programs. For example, the tool can be used to extract shellcode from malicious *Microsoft Office* documents or shellcode that is contained in network traffic. One of its limitations is that it does not become active *before* the execution of the first shellcode instruction. As an effect, no information can be gained about the exploited vulnerability that led to the execution of malicious code. One way to gain more insight about the actual root cause of the exploitation is to utilize BT in combination with this tool. This enables to virtually "look into the past" once the shellcode execution is detected: the concrete execution path that led to the malicious instruction can be reconstructed and examined in detail.

For the evaluation, we have added BT support to *CWXDetector* and examined a set of 4,869 malicious PDF documents. This malware corpus was originally collected by a well-known AV vendor in January 2011 from different sources [45] and it is known that each file in this corpus exploits some kind of vulnerability in *Acrobat Reader 9.00*. Hence, we could be sure that opening each file within that particular *Acrobat Reader* version would lead to a successful exploit. What we got as result of this experiment is a set of 4,869 different exploit reports with BT logs that cover the last 10,000 branches taken before the first shellcode instruction was executed. Note that we could also generate similar reports with other kinds of analysis tools (e.g., *Ether* or single stepping), but the performance overhead of BT is significantly smaller.

Example.

An example of a BT log excerpt is shown in Listing 1. The trace shows the behavior observed during the analysis phase based on the recorded branches. As discussed in Section 4,

for each branch we obtain a log message that contains the branch source code location and the branch target. In between these branches, we do not obtain any direct insights into what code was executed within a basic block. Nevertheless, this coarse overview of the branches taken by a program already contains enough information for our purposes as we demonstrate in the following.

```
[1704] from   0x781804d7   (MSVCR80.strcat+0x87)
[1704]   to   0x781804de   (MSVCR80.strcat+0x8e)
[1703] from   0x781804f6   (MSVCR80.strcat+0xa6)
[1703]   to   0x781804d9   (MSVCR80.strcat+0x89)
                1601 x     ....................
[ 101] from   0x781804f6   (MSVCR80.strcat+0xa6)
[ 101]   to   0x781804d9   (MSVCR80.strcat+0x89)
[ 100] from   0x80541f57   (ntkrnlpa.KiExceptionExit+0xab)
[ 100]   to   0x7c91e45c   (ntdll.KiUserExceptionDispatcher
      )
[  99] from   0x7c91e465   (ntdll.KiUserEx...Dispatcher+0x9
      )
[  99]   to   0x7c93a950   (ntdll.RtlDispatchException)
```

Listing 1: Excerpt of a Branch Trace generated for a malicious PDF file

Binning Approach.

Based on the collected data, we then clustered the BT reports into distinct bins, one for each exploited vulnerability. In order to achieve reasonable results for the binning, the logged data first has to be normalized in several ways. First, we used the relative addresses instead of the absolute ones, since the base address of modules could change over time due to techniques such as *Address Space Layout Randomization* (ASLR) [5,38]. Second, we collapsed loops, since we did not want to assign different bins to files that only differ in the number of loop iterations (e.g., due to differences in the size of the input data). To this end, we implemented an altered version of the approach from Tubella and Gonzalez [42], which main concept is that every backward jump forms a loop. Third, we also removed those parts from the traces that are related to the internal exception handling routines of the Windows system. Exceptions are frequently used by exploits in the last stage before control is transferred to the actual shellcode. By removing these parts from the traces, we prevent different exploits to be mistakenly put in the same bin because of this effect. Finally, we ignored those BTs of the actual executed shellcode since for binning these are not related to the vulnerability that was exploited. The shellcode is stilled logged and can be analyzed separately.

As clustering algorithm we have used *DBSCAN* [12] and as distance function a modified version of the *Jaro-Winkler distance* [19,48]. This function originally is used to measure the difference between two strings and calculates a similarity score based on several conditions. Mainly it is influenced by the amount of common characters and the amount of transpositions between them. Additionally, it prioritizes the prefixes of the compared strings, i.e., strings with a similar prefix get a higher score than those with only a similar suffix. This reflects our observation that branch traces (that are always considered backwards) require a common prefix if they reflect the same vulnerability. Experiments have shown that best results could be achieved if we prioritize the last 50 branches. For performance reasons we have further limited the overall amount of considered branches to 80. Preliminary tests have shown that nearly all characteristic variations of BT logs happen within these first 80 branches.

The DBSCAN algorithm has two configuration parameters that influence its behavior and quality: the minimum

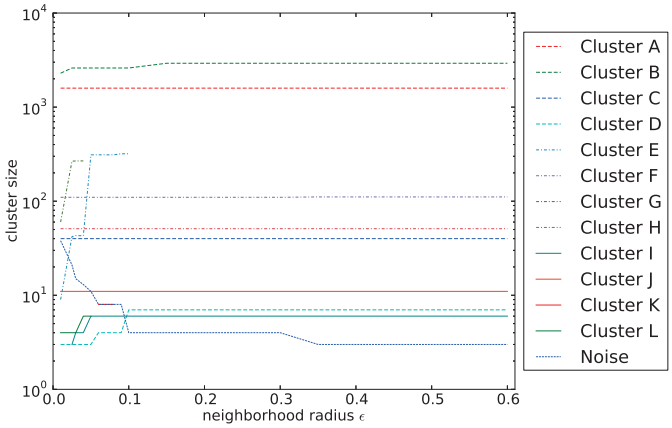


Figure 3: Distribution of Clusters Depending on ϵ.

cluster size k, which discriminates noise from valid cluster objects, and the neighborhood radius ϵ, that specifies the maximum distance of two objects to belong to the same cluster. The choice of k is merely a matter of taste (as long as it is greater than 1) and can be used to control the size of the resulting *noise*-cluster. Experiments have shown that $k = 3$ produces best results. In contrast, the value of ϵ has a severe effect on the number of resulting clusters, as can be seen in Figure 3. For very low values we get 12 different clusters and by increasing ϵ some of them merge into combined ones. From a value of 0.1 on we are left with 8 clusters and further increasing the neighborhood radius has no more effect on its number. However, for higher values the amount of the *noise* objects still decreases, because some of them fall into existing clusters. Note that the logarithmic scaling of the figure conceals that growing element size of these clusters.

We have manually analyzed randomly picked objects from the merging clusters and in all cases determined that they are based on the same root cause that is simply exploited in a different manner. For example there was a buffer overflow in a stack variable and one exploit diverts the control flow when the *memcpy* function is called with that buffer and another when *strcat* is executed. It is debatable if this is the same vulnerability or not. Nevertheless, by tweaking the ϵ radius one can determine how these cases are treated by the clustering algorithm. For the following comparison with other tools we have chosen $\epsilon = 0.1$, resulting in 8 different clusters and less than 10 outliers in the noise group.

Comparison With Other Approaches.

For further evaluation of our results, we compared them against those from the PDF analysis framework *Wepawet* [8]. This tool combines machine learning techniques with emulation and uses signatures of known CVEs to classify malicious documents and labels them appropriately. Note that many PDF documents do not only exploit one single vulnerability. Instead, they trigger different exploits, depending on the used PDF viewer application. For those samples Wepawet may not only generate one single label, but instead output a list of several CVEs. Furthermore, sometimes no known exploit can be detected at all and no label is generated. We have analyzed each sample with Wepawet, which resulted in *seven* different detected vulnerability signatures (CVE-2007-5659, CVE-2010-2883, SA33901, CVE-

2009-0927, CVE-2009-4324, CVE-2008-2992, CVE-2010-0188). After removing those CVEs from the resulting list, which only address exploits of Acrobat Reader versions other than 9.00, we are left with only *five* vulnerabilities (CVE-2010-2883, SA33901, CVE-2009-0927, CVE-2010-0188, CVE-2009-4324).

While most of our clusters have been consistent with the Wepawet results, we found two general differences. First, there was a small number of samples for which Wepawet did not return any CVE number (at least after removing the CVEs that do not affect the used Acrobat version). In contrast, our BT approach was able to successfully cluster those samples into six different clusters. We have manually verified a subset of those samples and learned that other samples from the same cluster seem to exploit the same vulnerability. Second, there have been some outliers that Wepawet detected as being malicious but labeled incorrectly. Again, we were able to manually verify that our clustering has grouped them correctly with other samples that exploited the same vulnerability.

Performance Evaluation.

Obviously, there is a performance impact when using BT. Nevertheless, all of the analyzed PDF documents actually executed their shellcode within a reasonable time: we set an upper limit for each analysis run of ten minutes and all runs finished in this time. More specifically, the fastest analysis took 11 seconds with BT (only 2s without BT), the slowest took 406s (117s without BT), and the average time was 129s (11s without BT). These measurements show that we have encountered a performance degradation factor of around 12 compared to the same system without BT. Apparently, this is magnitudes faster than performing single stepping on a native machine or with the help of hardware-assisted virtualization [10].

Discussion.

Our clustering approach and its evaluation have some limitations that we need to discuss. First, we cannot preclude that two *different* vulnerabilities are merged due to the fact that some function pointer is preliminary overwritten by different means, but then later called from the *same* calling site. If the number of executed branches between modifying the pointer and calling it exceeds a certain amount, our approach is blinded. Second, though we are using a sophisticated loop detection mechanism that is also able to collapse nested loops with varying loop iterations, it may fail to successfully collapse in certain situations. Finally, the biggest problem arises from the missing known truth about our samples. We are not able to manually analyze thousands of malicious PDF documents and there are no sources capable of delivering trustworthy information about the contained exploit, especially not AV products and other heuristic-based scanners. Therefore, we can only provide accurately generated evaluation data and reason about their validity. However, by comparing our results to those from *Wepawet* and further manually analyzing selected samples from our set, we are confident that our approach works properly.

5.2 Experiment 2: Enriching BT Logs

The aforementioned approach utilizes the BT log data in a straightforward way (i.e., by comparing the instruction addresses of different crashes). The derivable amount of information can be highly enriched by disassembling the particular instructions that are located at the branch sources and destinations. This enriched data can for example be used to detect code related to return-oriented programming (ROP) [37] and reconstruct the performed instruction sequences. Listing 2 shows an example of a BT log enriched with this kind of information where all *RET* and *CALL* branches are marked. If a *RET* without a corresponding `CALL` is detected, it is labeled as *ROP-RET* and this heuristic enables us to generically detect ROP code [9]. In the example, we can easily spot how the ROP code abuses existing, legitimate code chunks to prepare and actually perform calls to the API function *CreateFileMappingA* and *MapViewOfSection*.

```
[54] from 0x20c9ba54  AcroForm.DllUnregisterServer+0
     x485fa7
          (ROP-)RET  ####################
[54]   to 0x4a801f90  icucnv36.ubidi_getDirection_3_6+0x18
[53] from 0x80541f57  ntkrnlpa.KiExceptionExit+0xab
[53]   to 0x4a801f90  icucnv36.ubidi_getDirection_3_6+0x18
[52] from 0x4a801f91  icucnv36.ubidi_getDirection_3_6+0x19
          (ROP-)RET  ####################
[52]   to 0x4a807e7d  icucnv36.uhash_deleteUVector_3_6+0xc
[50] from 0x4a807e7d  icucnv36.uhash_deleteUVector_3_6+0xc
              CALL  --------------------
[50]   to 0x7c8094ee  kernel32.CreateFileMappingA
....
[26] from 0x7c809545  kernel32.CreateFileMappingA+0x70
               RET  --------------------
[26]   to 0x4a807e7f  icucnv36.uhash_deleteUVector_3_6+0xe
[25] from 0x4a807e7f  icucnv36.uhash_deleteUVector_3_6+0xe
          (ROP-)RET  ####################
[25]   to 0x4a801063  icucnv36+0x1063
[24] from 0x4a801064  icucnv36+0x1064
               RET  --------------------
[24]   to 0x4a8013df  icucnv36.ubidi_getReord...3_6+0x2aa
[23] from 0x4a8013e0  icucnv36.ubidi_getReord...3_6+0x2ab
          (ROP-)RET  ####################
[23]   to 0x4a8063a5  icucnv36.uenum_count_3_6+0x1d
[22] from 0x4a8063a6  icucnv36.uenum_count_3_6+0x1e
....
[20]   to 0x4a807e7d  icucnv36.uhash_deleteUVector_3_6+0xc
[19] from 0x4a807e7d  icucnv36.uhash_deleteUVector_3_6+0xc
              CALL  --------------------
[19]   to 0x7c80b995  kernel32.MapViewOfFile
```

Listing 2: BT log excerpt for ROP shellcode

As Listing 2 shows, the BT log is not only enriched by the CALL and RET sites, but our tool for branch tracing also takes advantage of the publicly available debug symbols from *Microsoft*. Obviously, for files from other vendors, these symbols are typically not available and, hence, it is more complicated to understand the semantics of the called functions and to isolate the root cause of a given vulnerability. Nevertheless, if such a tool is assisted by the (private) debug symbols, it is very easy to identify the actual code locations.

Though not explicitly mentioned, we already applied this kind of ROP detection during our first experiment described above. Since we did not want to take the branches of the actual shellcode into account, we had thus removed all branches that occured before the first ROP call. In total, we found 1,721 samples in our corpus that utilized an initial ROP stage and 3,148 which did not.

5.3 Experiment 3: Practical Delusion Attack with a PDF File

Finally we demonstrate a delusion attack with the help of a malicous PDF document. For that purpose, we have generated a specially crafted file that utilizes one of the delusion techniques introduced in Section 3. This document shows a different behavior when executed on a native machine compared to execution in a virtual environment like QEMU (and

as a result also in malware analysis frameworks such as *BitBlaze* [40] or *Anubis* [3]).

We used *Metasploit* [33] to create the PDF document. For exploiting the *CVE 2010-0188* vulnerability of the *Adobe Acrobat Reader 9.00*, we chose the *exploit/windows/fileformat/adobe_libtiff* module. We then created a modified version of the existing payload module *windows/messagebox*, in which we utilized the `rep movs` technique introduced in Section 3.3. The modifications of the payload module are listed in the extended paper version and the PDF document itself can be downloaded from *http://bit.ly/xtgRBE*.

We have analyzed this PDF document with the help of *Anubis* (which is based on QEMU) as well as with our BT approach. As expected, the malicious functionality (in our case just a simple message dialog) was only triggered when the document was opened on a real machine. Within the emulator, the PDF viewer simply closed and did not show any suspicious behavior. More precisely, it is not possible to achieve any insights into the malicious functionality since no such functionality is triggered at all. Note that even powerful analysis approaches like multi-path execution [27] cannot spot suspicious code regions since there is no explicit branch that is generally *not* taken during emulation. By using branch tracing, we were able to successfully observe and analyze the PDF shellcode.

6. LIMITATIONS

In this section we discuss the limitations of BT in general and of our specific prototype. First, the data obtainable by BT is rather coarse. Only the source and destination addresses of program branches and no information about runtime memory or register values can be gathered. Nevertheless, this is sufficient to reconstruct the complete execution path of an application. Section 5.1 demonstrates how this kind of information can be used to generate reasonable and useful analysis results. One way to increase the quality of the BT logs is to enrich them with disassembly information as we have shown in Section 5.2.

Another concern is the robustness of our approach against detection and evasion. Our current prototype could be detected by applying timing measurements to observe the introduced performance penalty. If the attacker is operating in ring-0, she is further able to deactivate or manipulate the BT settings directly. Besides deactivating the tracing feature there is no way to circumvent it, since the logging is done by the CPU itself. Nevertheless, these are drawbacks of our current implementation and could be addressed by incorporating a hardware-assisted hypervisor. With that help each read or write access of the related MSRs or the *time stamp counter* could be detected and simulated. However, incorporating external timing sources still allows to reveal the existence of BT, a general limitation that we share with all other automated analysis frameworks [3, 10, 40].

7. CONCLUSION

To obtain a detailed understanding of the behavior of exploits and malicious software, many different analysis techniques and frameworks have been developed in the past few years. A huge fraction of these systems is based on the utilization of software emulators since emulators enable a fine-grained control over the sample. As a result, attackers have constantly developed new methods and techniques to detect such analysis frameworks and armored their malicious programs appropriately. In this paper, we have presented new ways how an attacker can delude an emulator. Unlike other detection techniques, our methods do not combine an explicit environment check with a conditional branch. Instead they constitute *implicitly* different behavior on a native compared to an emulated machine caused by drawbacks of the particular operations and imperfections in the emulation process that cannot be mitigated easily.

This kind of delusion attacks motivates a new approach for dynamic code analysis: CPU-assisted *branch tracing*. This technique offers a granularity between instruction- and function-level monitoring and can be realized with reasonable performance overhead. In our view, the greatest advantage is the fact that the logging is performed by the processor itself and, hence, cannot be deluded since we obtain information about the actual executions performed by the CPU. In several practical experiments we showed that the obtained BT traces contain enough information to assist different tasks in malware analysis and vulnerability research.

Acknowledgement.

This work was supported by the Ministry of Economic Affairs and Energy of the State of North Rhine-Westphalia (grant 315-43-02/2-005-WFBO-009) and the German Federal Ministry of Education and Research (BMBF grant 01BY-1205A – JSAgents).

8. REFERENCES

[1] Advanced Micro Devices. AMD Lightweight Profiling Specification. Specification, AMD, 2010.

[2] Davide Balzarotti, Marco Cova, Christoph Karlberger, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Efficient Detection of Split Personalities in Malware. In *Network and Distributed System Security Symposium (NDSS)*, 2010.

[3] Ulrich Bayer, Andreas Moser, Christopher Kruegel, and Engin Kirda. Dynamic Analysis of Malicious Code. *Journal in Computer Virology*, 2(1):67–77, 2006.

[4] Fabrice Bellard. QEMU: A Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference*, 2005.

[5] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *USENIX Security Symposium*, 2003.

[6] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, and Randal E. Bryant. Semantics-Aware Malware Detection. In *IEEE Symposium on Security and Privacy*, 2005.

[7] Paolo Milani Comparetti, Guido Salvaneschi, Engin Kirda, Clemens Kolbitsch, Christopher Kruegel, and Stefano Zanero. Identifying Dormant Functionality in Malware Programs. In *IEEE Symposium on Security and Privacy*, 2010.

[8] Marco Cova, Christopher Kruegel, and Giovanni Vigna. Detection and Analysis of Drive-by-download Attacks and Malicious JavaScript Code. In *World Wide Web Conference (WWW)*, 2010.

[9] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. ROPdefender: A Detection Tool to Defend Against Return-oriented Programming Attacks. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2011.

[10] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: Malware Analysis via Hardware Virtualization Extensions. In *ACM Conference on Computer and Communications Security (CCS)*, 2008.

[11] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song. Dynamic Spyware Analysis. In *USENIX Annual Technical Conference*, 2007.

[12] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Conference on Knowledge Discovery and Data Mining (KDD)*, 1996.

[13] Peter Ferrie. Attacks on virtual machine emulators. `http://pferrie.tripod.com/papers/attacks.pdf`, 2007.

[14] Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin. Compatibility is Not Transparency: VMM Detection Myths and Realities. In *Workshop on Hot Topics in Operating Systems (HotOS-XI)*, 2007.

[15] Tal Garfinkel and Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Symposium on Network and Distributed System Security (NDSS)*, 2003.

[16] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based Whitebox Fuzzing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008.

[17] Intel Corporation. Intel: 64 and IA-32 Architectures Software Developer's Manual. Specification, Intel, 2007. `http://www.intel.com/products/processor/manuals/index.htm`.

[18] Intel Corporation. Intel Microarchitecture Codename Nehalem Performance Monitoring Unit Programming Guide (Nehalem Core PMU). Specification, Intel, 2010.

[19] Matthew A. Jaro. Unimatch: A record linkage system. `http://books.google.de/books?id=was9AAAAIAAJ`, 1978.

[20] Min Gyung Kang, Heng Yin, Steve Hanna, Stephen McCamant, and Dawn Song. Emulating Emulation-resistant Malware. In *ACM Workshop on Virtual Machine Security (VMSec)*, 2009.

[21] Martina Lindorfer Clemens Kolbitsch and Paolo Milani Comparetti. Detecting Environment-Sensitive Malware. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2011.

[22] Christopher Kruegel, William Robertson, and Giovanni Vigna. Detecting Kernel-Level Rootkits Through Binary Analysis. In *Annual Computer Security Applications Conference (ACSAC)*, 2004.

[23] Kevin P. Lawton. Bochs: A Portable PC Emulator for Unix/X. *Linux J.*, 1996, September 1996.

[24] Cullen Linn and Saumya Debray. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *Conference on Computer and Communications Security (CCS)*, 2003.

[25] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. Testing CPU emulators. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2009.

[26] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. Testing System Virtual Machines. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2010.

[27] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring Multiple Execution Paths for Malware Analysis. In *IEEE Symposium on Security and Privacy*, 2007.

[28] James Newsome and Dawn Xiaodong Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Network and Distributed System Security Symposium (NDSS)*, 2005.

[29] Peter Oehlert. Violating Assumptions With Fuzzing. *Security Privacy, IEEE*, 3(2):58 – 62, 2005.

[30] Travis Ormandy. An Empirical Study into the Security Exposure to Hosts of Hostile Virtualized Environments. `http://taviso.decsystem.org/virtsec.pdf`.

[31] Igor Popov, Saumya Debray, and Gregory Andrews. Binary Obfuscation Using Signals. In *USENIX Security Symposium*, 2007.

[32] Thomas Raffetseder, Christopher Krügel, and Engin Kirda. Detecting System Emulators. In *Information Security Conference (ISC)*, 2007.

[33] Rapid7. The metasploit framework. `http://metasploit.com/`.

[34] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware. In *Annual Computer Security Applications Conference (ACSAC)*, 2006.

[35] Joanna Rutkowska. Red Pill... or how to detect VMM using (almost) one CPU instruction. `http://invisiblethings.org/papers/redpill.html`, 2004.

[36] Hex Rays SA. IDA Pro Disassembler and Debugger. `http://www.hex-rays.com/idapro/`.

[37] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[38] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the Effectiveness of Address-Space Randomization. In *ACM Conference on Computer and Communications Security (CCS)*, 2004.

[39] Monirul I. Sharif, Andrea Lanzi, Jonathon T. Giffin, and Wenke Lee. Impeding Malware Analysis Using Conditional Code Obfuscation. In *Network and Distributed System Security Symposium (NDSS)*, 2008.

[40] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, Newsome James, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *International Conference on Information Systems Security (ICISS)*, 2008.

[41] Symantec. Internet Security Threat Report, 2010.

[42] Jordi Tubella and Antonio Gonzalez. Control Speculation in Multithreaded Processors through Dynamic Loop Detection. In *International Symposium on High-Performance Computer Architecture*, 1998.

[43] Amit Vasudevan, Ning Qu, and Adrian Perrig. XTRec: Secure Real-Time Execution Trace Recording on Commodity Platforms. In *Hawaii International Conference on System Sciences (HICSS)*, 2011.

[44] Amit Vasudevan and Ramesh Yerraballi. Cobra: Fine-grained Malware Analysis Using Stealth Localized-executions. In *IEEE Symposium on Security and Privacy, Oakland*, 2006.

[45] Carsten Willems. Using Memory Management to Detect and Extract Illegitimate Code for Malware Analysis. Technical Report TR-2011-002, University of Mannheim, 2011.

[46] Carsten Willems, Thorsten Holz, and Felix C. Freiling. Toward Automated Dynamic Malware Analysis Using CWSandbox. *IEEE Security and Privacy*, 5(2), 2007.

[47] Carsten Willems, Ralf Hund, Andreas Fobian, Dennis Felsch, Amit Vasudevan, and Thorsten Holz. Down to the bare metal: Using processor features for binary analysis. Technical Report TR-HGI-2012-001, Horst Görtz Institute for IT-Security (HGI), 2012.

[48] William E. Winkler. String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage. In *Proceedings of the Survey Research Methods Section*, pages 354–359, 1990.

Augmenting Vulnerability Analysis of Binary Code

Sean Heelan
sean.heelan@gmail.com *

Agustin Gianni
agustin.gianni@gmail.com *

ABSTRACT

Discovering and understanding security vulnerabilities in complex, binary code can be a difficult and time consuming problem. While there has been notable progress in the development of automatic solutions for vulnerability detection, manual analysis remains a necessary component of any binary auditing task. In this paper we present an approach based on run time data tracking that works to narrow down the attack surface of an application and prioritize code regions for manual analysis. By supporting arbitrary data sources and sinks we can track the spread of direct and indirect attacker influence throughout a program. Alerts are generated once this influence reaches potentially sensitive code and the results are post-processed, prioritized, and integrated into common reverse engineering tools. The data recorded is used to inform the decisions of users, rather than replace them. By avoiding the processing required for semantic analysis and automated reasoning our approach is sufficiently fast to integrate into the normal work flow of manual vulnerability detection.

1. MOTIVATION

Much of the commodity, commercial and industrial software in use today is only available in binary form. As a result, analysis of this software requires reverse engineering — a process that is primarily manual and usually labor and time intensive. Reverse engineering is a necessary step in vulnerability detection, malware analysis, crash triaging, exploit development, protocol recovery, interoperability engineering and other processes. Although the end goal differs, these tasks typically involve solving similar problems during the process of understanding a program in binary form. Due to the time and labor cost of reverse engineering, algorithms and techniques for assisting in the process are valuable.

In this paper we will focus on the goal of vulnerability detection, an area that has seen significant research interest over the past few years in response to the need for greater machine assistance. Progress has been made in the testing of file parsers [11, 12, 19], and system tools [7, 6] through techniques based on symbolic and concolic execution. However, many vulnerability types, software architectures and programs above a certain size are currently not handled by automatic methods. As such, a reverse engineer is required to manually assess the software. When one considers the task of assisting the reverse engineer in this process many new opportunities and research problems arise.

*Parts of this work were completed while the author was an employee of Immunity Inc.

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

2. INTRODUCTION

Over the past decade there has been extensive work in developing tools and theory based on formal program analysis with the goal of analyzing binary code. Among the most successful outcomes of this research have been those approaches based on symbolic/concolic execution. Combining binary instrumentation, software emulators and SMT solvers, automated solutions have been proposed for bug finding [11, 12], exploit generation [4, 1, 13], protocol reverse engineering [5], driver reconstruction [8], type recovery [21] and a variety of other tasks. These efforts have also resulted in the availability of several frameworks for binary analysis, including TEMU [22], BAP [3] and S2E [9].

The problems these tools tackle are faced daily by professionals involved in reverse engineering, vulnerability detection and exploit development. However, in general the solutions presented have not permeated into this industry. One possible reason is that the tools often do not integrate well with the work flow of the reverse engineer. Reverse engineering is generally a process of iterative, gradual understanding through forming hypotheses and looking for supporting evidence. Tools that have excessively long running times or do not support bidirectional information flow between the user and the tool are incompatible with such a process.

For example, frameworks that rely on full semantic emulation of instructions tend to have running times measured in hours for complex software like web browsers, document viewers and network servers. While such information may be necessary if one wants to automatically determine whether a program trace potentially contains a bug, we will later show how a more lightweight approach can provide a user with the required information to make this determination in a much smaller amount of time. As a result, more code can be covered while at the same time gaining an insight into the program that is not available if one relies on entirely automatic methods.

Another problem for many tools is that they are designed

to be entirely automatic solutions and as a result often lack the flexibility to deal with corner cases not envisaged by their designers. While reverse engineering involves many repeated patterns of work there are inevitably corner cases and complications in the analysis of every application.

Automatic approaches to bug finding also tend to be quite limited in the types of properties they can reason about, and as a result the types of bugs they can find. Tools such as KLEE [6] and EXE [7] can easily model and reason about arithmetic constraints and thus are targeted at bugs directly resulting from integer overflows, underflows and incorrect bounds. On the other hand they know nothing about dangling pointers, race conditions, type confusion and other common bug classes.

While full automation is desirable in many situations, in this paper we discuss an approach that is user centric. Instead of aiming to replace a user, we present a combination of dynamic taint analysis and lightweight static analysis to inform their decisions and increase efficiency during manual program analysis. Our algorithms focus on data gathering, analysis and display while leaving the specifics of problem solving up to the user.

We will direct the system towards the problem of manual vulnerability discovery but the information generated can also be used to seed more automated systems. A common problem faced by automated systems is in deciding what parts of the program to explore and how to explore them. The output we generate can be used to guide these decisions and target testing tools towards more interesting parts of the attack surface. As an example, similar technology has been successfully applied to provide guidance during fuzzing [10].

In section 3 we describe a system with support for direct and indirect data tainting. The system supports arbitrary data origins and sinks and can generate alerts based on function and instruction level information. These alerts focus the user on potentially sensitive areas of the code that are influenced by attacker controlled input. The primary goal of this tool is to assist in attack surface discovery and to prioritize regions of the code for assessment. Later in section 4 an evaluation of this tool on several real world applications.

Benefits of this approach include:

Generality Dynamic data-flow tracking is a well studied area [20] and usually works regardless of the software being analyzed. Instead of attempting to perform automated reasoning on top of this we instead focus on prioritizing the results of this analysis for human assessment. This allows the user to target the tool at software not generally handled by current automatic approaches such as network servers, web browsers and interpreters. It also assists the user when looking for bug classes that are not well handled by approaches based on symbolic execution e.g. use-after-free bugs.

Efficiency A key component of our approach is post-processing of the recorded data to allow the user to find the events they may be most interested in quickly. We take two approaches to this. The first is to use lightweight static analysis algorithms and a ranking function to prioritize particular results. The second is to extract relevant properties from recorded events and present them to the user in order to inform their decision making.

Speed We perform minimal analysis at run-time and so the overhead on the instrumented program is low. The time taken for post-processing of data is usually a few seconds. This is important because it allows the user to iteratively run the tool, view results, modify the configuration and re-run the tool without excessive wait times.

```
struct TaintInformation {
    size_t source_pc;
}

struct DirectTI : public TaintInformation {
    Descriptor      d;
    DescriptorInfo  di;
    Offset          o;
}

struct IndirectTI : public TaintInformation {
    vector <TaintInformation*> parents;
}

struct CompoundTI : public TaintInformation {
    vector<TaintInformation*> components;
}

struct TraceTI : public TaintInformation {
    BigInt sequence_num;
    vector<TaintInformation*> parents;
}

struct FlagUpdateTI : public TaintInformation {
    vector<TaintInformation*> parents;
}
```

Figure 1: Pseudo-C++ Taint Information Definitions. When we refer to instances of the `TaintInformation` class we include instances of all subclasses.

3. SYSTEM DESCRIPTION

`PINNACLE` consists of three high level components — a data tracking tool (`DT`) that monitors data as as it is processed by an application and generates alerts on events of note; a collection of static analysis scripts (`SA`) that process these alerts and the user interface (`UI`) that is integrated with `IDA` [14].

3.1 Data Tracking and Alerts

`DT` is implemented as a shared library on top of the PIN binary instrumentation framework [17]. We define `M` as the set of valid program memory addresses and `R` as the set of valid CPU register identifiers. For simplicity, in the remainder of this paper we will refer to the set of identifiers given by $M \cup R$ as `I`.

For each flag of the CPU's flags register we define a separate identifier in the set `R`, e.g. `CF` to identify the carry flag. We also define a separate identifier for each 8, 16 or 32 bit subregister. As an example, the various components of the `RAX` register can be referenced through `RAX`, `EAX`, `AX`, `AH` and `AL`.

A shadow data store `S` is maintained to keep metadata about each register or byte of program memory. We can consider `S` to work as a standard map from elements of `I` to instances of `TaintInformation`. Subclasses of the `TaintInformation` class can differ in terms of their attributes depending on the source of that taint information and the taint propagation mechanism in use. The details of this class will

be elaborated on in section 3.1.1, while its definition is provided in Figure 1.

On starting a program, or attaching to one, the following holds:

$$\forall x \in I : S[x] = NULL$$

A memory location or register $x \in I$ is defined as *tainted* if $S[x] \neq NULL$.

`DT` is concerned with introducing tainted data, propagating it across memory locations and registers, and generating alerts when an instruction in combination with the metadata in `S` matches against a defined set of rules.

3.1.1 Introducing Tainted Data

`DT` uses the instrumentation mechanisms provided by PIN to update the shadow data store S. PIN allows one to instrument at varying levels of granularity but, for the purposes of introducing new tainted data, we instrument at the function level. With compiled code we need not limit ourselves to higher level function boundaries but can instead consider any arbitrary chunk of assembly code to be a function. However, practically speaking we will choose code that corresponds to a traditional higher level function as this is usually the level of granularity we require when considering the introduction of tainted data.

For any function f that we consider to introduce tainted data we define a pair of functions (f_{pre}, f_{post}) that work together to update S with new taint information. f_{pre} is run before the execution of f and can access the parameters of f. f_{post} is run after the execution of f and can access its return value and any other outputs. For each location $x \in I$ that f updates with attacker controlled data, (f_{pre}, f_{post}) will create a new metadata instance d that is a subclass of `TaintInformation` and update S via $S[x] = d$. This indicates that the memory location or register identified by x is tainted and metadata is provided by d.

During vulnerability discovery the reason for marking a location as tainted is typically to identify it as being influenced by an attacker and then to determine how that influence may effect the rest of the program's control and data flow. *Influence* is a matter of degrees however [18] and the level of control one may have over a value in memory or registers can vary. In `DT` we reflect this fact by allowing for data to be marked as either directly or indirectly tainted.

For each $d \in (d_0, \ldots, d_n)$ created by (f_{pre}, f_{post}) a subclass of `TaintInformation` must be selected. Two subclasses are available, `DirectTI` and `IndirectTI`, which can be seen in Figure 1. We will say that a location $x \in I$ is *directly tainted* if $S[x]$ is an instance of `DirectTI` and *indirectly tainted* if $S[x]$ is an instance of `IndirectTI`.

The function pair (f_{pre}, f_{post}) are manually created so a developer/user of `DT` can use their own judgement as to the type used for each $d \in (d_0, \ldots, d_n)$. As we do not perform any automatic input generation based on the taint information the correctness of the chosen type is not critical. However, the type of influence an attacker has is used in prioritizing results and thus it does have some importance.

`DirectTI` is used when we consider the data stored at the register or memory location identified by $x \in I$ to be directly controlled by the attacker. For example, when a value is read into a program unconstrained and unchanged from an attacker controlled source, such as a socket or file descriptor, we create a `DirectTI` instance d for each written memory address x and then set $S[x] = d$. Direct sources are typically functions such as *read*, *recv* and so on.

$(w_0, b_0) \leftarrow \{(r_0, b_0), (r_1, b_0)\}$
$(w_0, b_1) \leftarrow \{(r_0, b_1), (r_1, b_1), (r_0, b_0), (r_1, b_0)\}$
$(w_0, b_2) \leftarrow \{(r_0, b_2), (r_1, b_2), (r_0, b_1), (r_1, b_1)\}$
$(w_0, b_3) \leftarrow \{(r_0, b_3), (r_1, b_3), (r_0, b_2), (r_1, b_2)\}$
$flags = (ZF, CF, OF, PF, AF, SF)$

Figure 2: Syntactic specification for the *add reg32, mem32* instruction

Alternatively, we may use `IndirectTI` when we consider the data stored at $x \in I$ to be indirectly under the control of an attacker. Indirect sources of control are more diverse and sometimes less obvious than their direct counterparts. Often indirect influence arises when an output from a function has a control flow dependency on a location $x \in I$ for which $S[x] \neq NULL$. A straightforward example of this form of indirect influence is the *strlen* function. If a program contains a call to *strlen(s)* where s is controlled by an attacker then the return value is certainly influenced, albeit indirectly. For such cases we can connect the new `IndirectTI` objects to the `TaintInformation` objects of the data which induced the control flow dependency.

For other sources of indirect influence there may be no clear data or control flow dependency between the updated location $x \in I$ and another location $y \in I$ for which $S[y] \neq NULL$. For instance, the return value of the `recv` function can be indirectly controlled by an attacker based on the number of bytes sent and also by closing the connection. For such cases there is no parent `TaintInformation` instance to connect the new `IndirectTI` instance to but as we are not aiming to automatically generate new inputs this is not problematic.

3.1.2 Propagating Taint Information

After the first execution of a (f_{pre}, f_{post}) pair that updates S such that $(\exists x \in I \mid S[x] \neq NULL)$ is true the taint propagation mechanisms of `DT` are enabled. From that point onwards every instruction executed is hooked to detect if it operates on bytes that are tainted and, if so, to propagate that taint information to any written registers, flags or memory locations. When this process starts a global tainted instruction counter is initialized to zero and incremented on each instruction that propagates tainted data. This is simply used later to assist in reconstructing traces.

For each instruction we define the set of outputs $U \subset I$ as the registers, flags or memory locations that are modified by the instruction. On the *x86/x64* architectures each instruction may have multiple outputs. Each output $o \in U$ is a function over a set of inputs $N \subset I$, where N may be the empty set $\emptyset$. If we define the set of tainted inputs T as $\{i \in N \mid S[i] \neq NULL\}$ then *taint propagation* occurs when, for a given output o, $T \neq \emptyset$. When these conditions are met we propagate the information from T to o in S.

Before discussing propagation we must first mention how multi-byte operations are processed to produce the required relationship between inputs and outputs. For each *x86/x64* instruction that we handle we provide a map M that describes the relationships between inputs and outputs in terms of the instruction's operands. This is a syntactic description of the relation from inputs to outputs with no semantic information. For example, for the *add reg32, mem32* instruction

the map looks as described in Figure 2.

(w_n, b_m) stands for the m^{th} byte of the n^{th} write operand and (r_n, b_m) stands for the m^{th} byte of the n^{th} read operand. If we inspect the mapping for the second output byte we can see that it defines the output as a factor of both the second bytes of both inputs *and* the first bytes of both inputs. This is to account for potential carrys in the addition. For instructions like *shl reg32, cl*, where we do not know exactly which inputs may contribute the which output, we over-approximate. Typically this is done by saying each output $o \in U$ is a factor all elements of N. The map also defines the flags that may be updated by the instruction.

We begin propagation by converting each $(op_n, byte_m)$ pair to an identifier $i \in I$. For registers this is done by means of another map which, for any register operand and byte offset, returns the correct identifier used to reference that particular subregister, or byte. For memory addresses we can simply compute the identifier by adding the byte offset to the base address. By iterating over M we can then produce a set of tuples $R = (o \in U, N \subset I)$ defining the byte level relationships between inputs and outputs. This provides us with the required information to update $S[o]$ for each tuple.

Input Tracking.

`DT` allows for two different taint propagation modes. In the first mode, which we will call the *tracking mode*, each instruction simply spreads the taint status of inputs through to outputs without recording the instruction address, or any other identifying information, responsible for the taint propagation. In this mode, for any given location $(i \in I \mid S[i] \neq NULL)$ we can tell which program input bytes may have contributed to the value in location i. However, we cannot tell the instructions that were responsible for manipulating and moving that tainted data from the program input to location i.

For each tuple $(o \in U, N \subset I) \in R$ we begin by retrieving the corresponding `TaintInformation` instances for each input $i \in N$, producing a set of `TaintInformation` instances V. It is possible that an element of V is in fact a `CompoundTI` instance, representing the join of influence from multiple input bytes. If this is the case we recursively replace that `CompoundTI` instance with the members of its `components` attribute. This is done to ensure that `CompoundTI` instances that are created, used to influence a new location and then overwritten can be safely deleted, thus minimizing the amount of memory required. Once this has been performed for all elements of V the final set consists only of `DirectTI` and `IndirectTI` instances.

If V contains two or more elements we create a `CompoundTI` instance and set its `components` attribute equal to a vector constructed from V. The end result is that every element of S is either a `DirectTI`/`IndirectTI` instance or a `CompoundTI` instance with a `components` attribute consisting of instances of these types.

Input Tracing.

In the second mode, which we will call the *tracing mode*, each instruction creates new `TraceTI` instances for each output, allowing us to track exactly which instructions were involved in the computation of any given byte of tainted data. Under this mode, for any given location $(i \in I \mid S[i] \neq NULL)$ we can construct a tree by recursively traversing the parents of each `TraceTI` instance. The leaves of this tree will be `DirectTI`/`IndirectTI` instances and from it we can present the user with a list of all instructions involved in the computation of i.

To propagate the taint information we first iterate over the tuples $(o \in U, N \subset I) \in R$ retrieving the corresponding `TaintInformation` instances for each input $i \in N$, producing a set of `TaintInformation` instances V. In this mode, an element of V may be a `DirectTI`, `IndirectTI` or `TraceTI` instance. The key difference is that we do not expand `TraceTI` instances like we did `CompoundTI` instances. Instead we create a new `TraceTI` instance and set its `parents` attribute equal to a vector constructed directly from V. This ensures that for any element of S we have available the information to construct the tree linking it back to input data. The other difference is that `TraceTI` objects store the global tainted instruction counter in the `sequence_num` attribute. This is integral in later reconstructing the order in which updates were made to values that are not derived from one another. It is also used to align the individual byte updates performed in a single multi-byte instruction when processing traces.

Value Constraints.

The above processes allow us to track or trace data between a taint source and a taint sink. However, it is also desirable that we would be able to report on the *constraints* imposed on this data. For example, if the current path through the program contains a `test` instruction that operates on tainted data, followed by a conditional jump, then this may impose a restriction on the set of values that data may have and still trigger the same path. Such information is useful in differentiating safe from unsafe arithmetic performed by a program.

In order to facilitate this, we allow a user to enable an optional mode that propagates taint information to the CPU flags register. For each instruction that updates CPU flags and operates on tainted data we create a `FlagUpdateTI` instance. The `parents` attribute of this instance is set to a vector of all `TaintInformation` instances across all operands of that instruction. For all flag identifiers $i \in I$ updated by the given instruction, S is then updated by setting $S[i]$ equal to the `FlagUpdateTI` instance.

On a conditional jump instruction we then retrieve the `FlagUpdateTI` instance for each flag used and merge their `parents` vectors together into a new set C. C is a set of all `TaintInformation` instances that may affect the outcome of the conditional jump. A tuple (C, a, n) is then appended to a list used to represent the path condition, where a is the current instruction address and n is the current value of the global tainted instruction counter. This list is periodically flushed to disk.

The Performance/Accuracy Tradeoff.

The primary downside to a syntactic, rather than semantic, approach to data flow specification is related to accuracy. Without access to the semantics of an instruction it is possible to conclude that a location in S is tainted when it is not. As an example, consider the instruction `xor eax, eax` and assume that the EAX register is tainted. If the handler for `XOR` shares the specification of other bitwise operators, like `AND` and `OR`, we would incorrectly conclude that the EAX register is still tainted after this instruction. Taking the `XOR` of a location with itself in order to clear it is a common op-

eration however, and so we can specifically encode a handler for this situation to avoid an erroneous result.

In general, such a solution is not possible without bit-level taint tracking and semantic specifications for instructions. As a result, in some situations there will be elements of S that are incorrectly marked as tainted. The following code is a contrived example but demonstrates the problem:

```
0:  cmp al, 0xf0
1:  jne EXIT
2:  and al, 0x0f
```

Assume that the AL register is tainted at address `0`. The task is to decide at address `2` whether this still holds. At `2` the only information available to `DT` is that $S[AL] \neq NULL$ as there is no facility for reasoning about the semantics of the path condition to detect the implied boundary on the value of the AL register. We must therefore conclude that it is possible for at least one of the lower four bits of AL to be set and thus that $S[AL] \neq NULL$ still holds after the execution of the instruction.

While this issue is worth noting, in our experience it is not a common problem that interferes with the users confidence in the data presented. In the domain of concolic/symbolic execution over-tainting can still occur [15], albeit for different reasons. Similar to our work however, this is rarely a significant problem.

3.1.3 Tainted Data Sinks

`DT` allows for configurable and on-demand *sinks* to execute a handler once they detect tainted data. The purpose of these handlers is usually to inform the user once attacker influenced data reaches code that may be of interest. We provide handlers for common sinks e.g. memory allocation or data copying functions, like `malloc` and `sprintf`, used with tainted parameters; instructions with a `REP` prefix and a tainted ECX register; conditional jumps based on tainted flags and so on. However, as one reverse engineers an application it is common to want to set application-specific handlers. We support this by allowing users to specify an $(address, shared\_library)$ pair and inserting hooks from their shared library at the address provided. This allows monitoring of custom memory management routines, third party libraries, and any other code without modification to the core of `DT`.

Depending on the type of sink, the number of hooks required and their interactions may differ slightly but in general they operate as follows. For a given sink k we define a function pair (k_{check}, k_{alert}), where k_{check} inspects a set of program locations given by $L \subset I$ and k_{alert} logs sufficient information to allow prioritization of results by static analysis and/or, importing of the results into `IDA` for inspection by the user. Let $T = \{l \mid l \in L, S[l] \neq NULL\}$, then k_{alert} is executed if $T \neq \emptyset$. k_{alert} will typically log the address of the sink along with the information on all members of T. If the *tracking* propagation mode is in use then this information will be limited to the tainted inputs that influenced the members of T. However, if the *tracing* propagation mode is in use k_{alert} will have access to this information as well as a trace of every instruction that modified those input bytes between their introduction and the sink k. For each element of T the sequence of parents back to the taint source will be logged, including instruction addresses and the `sequence_num` attribute.

3.2 Post-Processing and User Interface

`DT` produces a variety of outputs containing information useful in the processes of vulnerability identification and reverse engineering. The following categories of information are made available:

A Program Trace A user can choose to log each instruction the first time it processes tainted data or, alternatively, a log of each instruction every time it processes tainted data.

Function Alerts Each (k_{check}, k_{alert}) pair as described above may result in a log of inputs, or inputs and then modifying instructions, every time that function is called with tainted parameters.

Instruction Alerts Certain instructions can also be hooked with a (k_{check}, k_{alert}) pair and produce the same style output as function hooks. Of particular interest are the execution of `REP` prefixed instructions with a tainted ECX register and conditional jumps that are influenced by tainted flags.

The Path Constraints If this feature was enabled then a log of the branch points conditional on user input, as well as the bytes they were influenced by will be available.

Applications that perform significant processing on attacker controlled inputs can result in large amounts of data being logged across these categories. Table 2 shows the number of unique instructions acting on tainted data with per tested application. These figures range from approximately 5000 instructions to upwards of 25,000. When working on smaller targets it may be sufficient to simply display this information in a table to the user. For larger targets however, such as those discussed in section 4, it is necessary to process, rank and then display this information in such a way as to focus the user on the most relevant results.

The processing and ranking functionality that makes up `SA` consists of several Python scripts that run on top of `IDA`. As input they take the results of `DT` and as output they integrate information on tainted data flow into the disassembly view provided by `IDA`, as well as extending it with new tables of relevant information. The analysis component `SA` and user interface components `UI` are quite closely linked so we will explain their features in tandem.

3.2.1 Attack Surface Identification

The first stage of reverse engineering driven vulnerability detection requires one to find the attack surface of a program. We will loosely define the attack surface as code with data-flow or control-flow that can be directly or indirectly influenced by an attacker. This stage is critical as typically one wants to find the most security flaws in the least amount of time and the ability to prioritize different parts of the code for review is crucial to this process.

Often applications come with their functionality spread through a number of shared libraries, each with potentially hundreds or even thousands of individual functions. For example, one of the applications discussed in section 4 contains 43 DLLs, several of which have more than a thousand functions. `PINNACLE` makes it easy to quickly discover which of these process attacker controlled input and which do not.

Previously, a popular approach to this problem has centered around a technique called *differential debugging* or *differential reverse engineering* [16, 23, 2]. Under this approach one attempts to discover the code responsible for particular features of a piece of software by recording the basic blocks executed when that functionality is not used and then again when it is used. By taking the difference between these two sets of basic blocks one usually ends up with a set of basic blocks that are related to the feature of interest. Differential debugging is a simple but relatively effective solution to the problem. However, using data tainting information we can get much more fine grained information. Instead of concluding that a collection of basic blocks may be somehow related to the feature in question we can conclude the precise instructions responsible for processing every byte of user input.

We break the process of attack surface identification down into two stages, firstly at the library level and then at the function level.

Library Processing.

Using the program trace we group the recorded instructions by DLL and count them. Following this we also group and count each function and instruction alert. The results are then displayed in a table to the user who can easily see those DLLs that perform the most work on attacker controlled inputs and also those that generate the most alerts for events of note.

Often applications will encapsulate functionality related to IO within a particular library and then tainted data will flow from this library through a number of other libraries before reaching a taint sink. If the tracing mode is enabled then on an alert we can create a graph visualizing this information flow for the user.

Attack Surface by Function.

Using the information provided by the *program trace* and the analysis API of `IDA` we update the disassembly view of `IDA` to highlight every instruction that has processed tainted data. A count per function of each alert generated by that function, its cyclomatic complexity, the number of instructions in the function and the number of tainted instructions is then placed in a sortable table and displayed. Any functions that introduce tainted data via a taint source are also highlighted.

The information displayed allows a user to get a feel for which functions perform the most processing and are potentially the most interesting locations to begin auditing. Currently we do not attempt to perform any automated ranking of functions but allow the user to select from the above metrics and sort the results.

3.2.2 Alert Prioritisation and Display

For each alert generated by a k_{alert} function the output will depend on the propagation mode in use. Under the tracking mode the output will be the information from a set of `DirectTI` and/or `IndirectTI` instances that allow us to identify a set of byte inputs from a file or network stream. Under the tracing mode the output is much more verbose and allows us to link the bytes of interest at the point of the k_{alert} execution to a set of input bytes through each instruction that may have transformed them along the way. The resulting information is a set of input bytes and a sequence of instruction addresses $A =< a_0, \ldots, a_n >$. Processing the produced log into this set of address is not discussed here due to space considerations. The process is relatively trivial and simply a matter of iterating over the produced log and building the set of instruction addresses, ordered by the tainted instruction counter.

For both the tracing and tracking mode we may also have access to the path condition. This can also be processed into a set of addresses $< c_0, \ldots, c_n >$ that identify tainted conditional branches. Assuming we are operating under the tracking mode the addresses of the conditional branches can be integrated into A using the associated tainted instruction counter values.

Due to the volume of alerts that may be generated it is necessary to have some form of prioritisation mechanism. We do this by performing a simple analysis of every instruction in A. If we detect an arithmetic operator like `add` the *has_arith* flag is set, if we detect a bitwise operator like `and` the *has_bitwise* flag is set and finally, if there are any conditional jumps in A the *has_jcc* flag is set. The attributes are ranked as follows:

`has_arith` > `has_bitwise` > `has_jcc`

The ranking of a trace is given by the least of the attributes assigned to it. When two traces have the same ranking, the one with the shorter sequence of addresses is ranked higher.

Naturally, the *has_arith* and *has_bitwise* attributes only apply if we are operating under the tracing mode. In the tracking mode we have no choice but to simply rank traces without conditional jumps higher than those with them.

The list of alerts are then displayed to the user via a table in IDA, where the columns contain the name of the alert followed by the above calculated attributes. Once an alert is selected another table is generated listing every instruction in A. The disassembly view is also updated to color each of the instructions listed in A.

4. USAGE AND EVALUATION

For the purposes of this paper we have evaluated `PINNACLE` on the targets listed in table 1. The targets are all complex, commercial software and available only in binary form [1]. The purpose of the applications are diverse, including one of the most frequently encountered image editing suites (*ImageSuite*); a multi-threaded game server that typically has 50,000 online players at any one time(*GameServer*); the desktop control center for a common brand of network printer (*PrinterControl*) and a popular industrial design suite (*InDesign*). Between them their functionality is scattered across several hundred DLLs and several hundred thousand functions.

Reverse engineering any of these targets for the purposes of vulnerability detection can be a time consuming process but in this section we will demonstrate how `PINNACLE` enables a user to quickly discover the program's attack surface and then audit that attack surface in a prioritized manner. Using `PINNACLE` we were able to discover security vulnerabilities in all targets in a short amount of time.

[1]The targets have been anonymised as the security flaws found are unpatched

4.1 Performance

The primary output of PINNACLE is intended for human consumption, and thus an evaluation of the quality of this information is subjective. An idea of the usefulness of this data is provided in section 4.2 where we go through a number of serious flaws found with PINNACLE. However, we can provide metrics on the run-time impact of PINNACLE during the instrumentation phase as well as the size of the identified attack surface relative to the overall quantity of code executed. It is important to measure the run-time impact of the tool as this is the most time consuming part of the analysis. As mentioned in the introduction, excessively long run-times may not integrate well into a user's work flow. By measuring the difference between unique executed instructions and the subset of these that process tainted data we can get an idea of whether the wait for this information is worthwhile, versus an alternative approach such as differential debugging.

Table 1 shows the run-time impact of PINNACLE in comparison to a basic block coverage tool, used for differential debugging. The experiments were performed on a 32-bit dual-core Intel 2.20Ghz processor with 3GB of RAM running Windows 7 [2]. The basic block coverage tool is taken as a base and and the results of other methods are a factor of that. The actual program run time in seconds is also provided. The tasks performed were to scan a document (*PrinterControl*), load an image from disk (*ImageSuite*), connect a client to the game (*GameServer*) and open a design specification (*IndDesign*).

We can see that, as expected, PINNACLE has a higher runtime impact than simple basic block coverage tracking. However, the run-time impact is not sufficiently large as to prevent the integration of the tool into a reverse engineer's work flow. Interestingly, in some cases the impact of running PINNACLE in *tracing* mode is not significantly higher than running in *tracking* mode. In these cases we can get the benefits associated with the extra information provided, without run-times that might be inconvenient to the user. The two exceptions were with *GameServer* and *IndDesign*.

While running the *GameServer* under the *tracing* mode the client displayed a connection timeout error after approximately 80 seconds. By this point the client and server had already exchanged a number of requests and responses so we were still able to gather information on a useful portion of the connection protocol. With *IndDesign* under tracing mode the system eventually ran into an out-of-memory (OOM) condition while loading the input. The *IndDesign* application performs extensive decompression on the input. As a result of this, the application performs significantly more processing of tainted data, in comparison to the others. This leads to an OOM condition in PINNACLE due to the large chains of objects necessary to maintain information on each instruction operating on tainted data. Despite this, we were able to find a number of security vulnerabilities using the information from the tracking mode alone. However, both this OOM and the timeout on *GameServer* show that there is still work to be done on optimizing the instrumentation phase of our approach.

[2] PINNACLE also works on Linux and x86-64

One of the most useful applications of PINNACLE is in attack surface identification. In table 2 we can see a comparison of the number of loaded DLLs, including system DLLS, versus the number of DLLS that actually process tainted data. We can clearly see that taint tracking allows us to significantly cut the number of DLLs we may consider interesting from an attackers perspective. A large number of DLLs may still remain if we only consider those that process tainted data, however, using the data flow graph mentioned in section 3.2.1, along with basic metrics on the number of tainted instructions per DLL, a user is provided with useful guidelines on where significant processing is performed.

Table 2 also provides metrics on the number of unique instruction addresses executed versus the number of those that process tainted data. PINNACLE cuts the attack surface down to a fraction of those instructions actually executed e.g. by a factor of 120 in the case of the *PrinterControl* application. This saves significant amounts of time for a reverse engineer and is a more accurate solution in comparison to differential debugging.

The ability to focus quickly on code that is influenced by an attacker, in combination with direct targeting towards potential vulnerabilities, means that a user can be more efficient than otherwise. We have tested this hypothesis by real world use of PINNACLE.

4.2 Vulnerability Discovery

4.2.1 *Remote Overflow in* PrinterControl

In the application *PrinterControl* we found several vulnerabilities that can lead to remote compromise of the device attempting to interact with the network printer. One such vulnerability is found in the initialization stages of the application. When *PrinterControl* starts it sends a broadcast request over the network to identify an available printer/scanner. Any device on the local network can see and respond to these requests. By starting *PrinterControl* under PINNACLE we were able to detect a serious vulnerability resulting from insecure parsing of responses to this initial broadcast.

Several function alerts related to memory allocation using a tainted size are generated during this initial phase. The majority are in a single DLL, which we will refer to as *Parse.dll*. Using the data flow graph available, we can see *Parse.dll* uses a second DLL *Net.dll* to read data from a network socket through an exported function *Net.dll!Read*. In *Parse.dll* there are 67 call sites to the allocation function used, which PINNACLE narrows down to 10 as being dependent on attacker provided input. In all cases the allocation size is first read from data received over the network, then that same size is provided to *Net.dll!Read* to fill the allocated buffer. Using the taint information from PINNACLE we can quickly inspect these call sites and conclude that all uses of allocated buffer and attacker influenced size are safe.

However, if we instead inspect *Parse.dll* by looking at call sites to *Net.dll!Read* where the size is tainted the results are more interesting. Under this view we can see a single call to *Net.dll!Read* where the size to be read is taken from a previous network packet and the destination buffer is the stack. PINNACLE can provide us with each check and modification performed on the size value, allowing us to confirm that any non-zero size can be provided.

Pseudo-code for the vulnerability can be seen in Figure 3. The flaw itself is trivial to spot once the code has been iden-

Application	BBCov	PINNACLE Track	PINNACLE Trace
PrinterControl	x1 (58s)	x4 (210s)	x5 (290s)
ImageSuite	x1 (71s)	x3 (240s)	x13 (945s)
GameServer	x1 (12s)	x6 (68s)	Timeout
IndDesign	x1 (615s)	x5 (3000s)	OOM

Table 1: Runtime impact of various approaches to attack surface identification and vulnerability discovery

Application	# Loaded DLLs	# Tainted DLLs	# Unique Exec. Instrs.	# Unique Tainted Instrs.
PrinterControl	104	30	712134	5908
ImageSuite	165	33	603922	25263
GameServer	59	18	242671	12575
IndDesign	289	55	2811259	94598

Table 2: Actual loaded DLLs and executed instructions versus those influenced by attacker input

```
int res;
size_t next_read;
char buf[12];

...

res = read_data(&buf, 12);
if (!res)
    return;

...

next_read = get_size(buf + 6);
res = read_data(&buf, next_read);

...
```

Figure 3: *PrinterControl* remote overflow vulnerability

tified and the work performed by the called functions is figured out. PINNACLE makes both of these processes easier by providing direction on where to look and data flow information that can give intuition as to the sources and sinks for tainted data. We also found that PINNACLE made it easier to verify the non-vulnerable uses of *Net.dll!Read*. By highlighting the relevant code in IDA less time needed to be spent on each case and thus we could move on to more interesting cases.

4.2.2 *Heap Overflow in* IndDesign

The *IndDesign* application uses a compressed file format that requires extensive processing to decompress and parse. Using PINNACLE we were able to quickly locate the decompression algorithms and then discover a heap overflow in a related function. The impact of this bug is that opening an attacker provided design can lead to arbitrary code execution.

The file format used consists of multiple segments and a table that provides a size and checksum for each segment. We identified the functions responsible for the integrity checks and decompression by searching the tainted instructions for conditionals based on the checksums located in this table. As PINNACLE provides the index into the file of each tainted byte processed by each instruction, this search can be automated using a small script. Immediately after the integrity checks the decompression takes place.

```
int process_segment(char *data, size_t data_sz)
{
    size_t decompressed_sz = *(size_t*) data;
    if (decompressed_sz > 0x1000)
        return -1;

    char *decompressed_data = malloc(decompressed_sz);
    decompress(data, data_sz, decompressed_data,
                decompressed_sz);
}

void decompress(char *compressed_data,
    size_t compressed_sz,
    char *decompressed_data,
    size_t decompressed_sz)
{
    char *cur_byte = compressed_data;
    size_t copy_counter = 0;

    while (*cur_byte++ == CONSTANT1)
        copy_counter += CONSTANT2;

    while (copy_counter > 0) {
        expand_data(cur_byte, &decompressed_data);
        cur_byte++;
        copy_counter--;
    }
}
```

Figure 4: *IndDesign* heap overflow vulnerability

```
...

recv_len = recvfrom(s, buf, len, flags, from,
            fromlen);

...

computed = compute_checksum(buf, recv_len);
sent = (*(char*)(buf + recv_len - 2)) << 8 |
            (*(char*)(buf + recv_len - 1))

if (computed != sent)
    checksum_err();
...
```

Figure 5: *GameServer* unchecked return value usage

An alert on a tainted argument to the `malloc` function directs our attention towards the `process_segment` function shown in Figure 4. By inspecting the data flow information we can see that `decompressed_sz` is a 32-bit field, taken from the attacker provided file. Its range is restricted to values between 0 and 0x1000 and a buffer is allocated to store the decompressed data based on this information.

By following the `decompress` call in `IDA` we can see, via the taint information, that the first `while` loop is bounded based on comparisons with attacker provided data. By controlling the input bytes to equal `CONSTANT1` we can manipulate `copy_counter` to hold values far larger than 0x1000. The loop calling `expand_data` is then bounded using the `copy_counter` variable instead of `decompressed_sz`. In this case it is the fact that the copy loop is *n*ot bounded by the same attacker controlled value used to allocate the destination buffer that is conspicuous.

By inspecting `expand_data` we can immediately tell, based on the data flow information, that it either copies attacker controlled bytes directly to the output buffer or decompresses them into a longer sequence. The end result of this is that we can manipulate the decompression algorithm to overflow the `decompressed_data` heap buffer with a controllable amount of data. Even with out the full tracing information available `PINNACLE` makes the process of tracking down such issues and understanding the surrounding code much easier than doing so using just a debugger, disassembler and coverage information.

4.2.3 *Unchecked Use of* recvfrom *Return Value in* GameServer

The *GameServer* application serves as a useful demonstration of a bug that is quickly uncovered when indirect tainting is used. The main processing loop of *GameServer* begins by reading data from a socket, then computes a checksum of this data and compares it against a value stored in the last two bytes of the received data.

Figure 5 shows pseudo-code for this initial function. Upon execution of this function the data read into `buf` will be marked as directly tainted while the value in `recv_len` will be marked as indirectly tainted. On the computation of the value of the `sent` variable `PINNACLE` creates an alert, if memory read/write based alerts are enabled. When the alerts are processed and loaded into `IDA` this will be assigned a high priority as a value indirectly controlled by an attacker is used, unchecked, in a memory dereference. If an attacker provides data of length 0 or 1 to the server the computation of `sent` will access 2 or 1 bytes *before* the `buf` buffer. This flaw will not result in malicious code execution but under certain circumstances could lead to a denial-of-service condition if `buf` is located within a byte of a page boundary and the previous page is unmapped.

While the flaw itself is not serious this example provides motivation for ensuring we are thorough in noting and tracking indirect sources of attacker influence.

5. CONCLUSION

Understanding software, especially software in binary form, can be both time consuming and difficult. However, it is also a necessary requirement in order to solve a diverse set of problems. In this paper we have presented techniques for attack surface discovery and vulnerability detection that are designed to increase efficiency in the work flow of a reverse engineer. Our results demonstrate the usefulness of attempting to solve *just enough* of a problem automatically, while relying on a human for creativity and decisions in the use of that data.

Much remains to be done in discovering the most effective ways to present users with the results of automated analysis and, likewise, on how best to extract their domain specific knowledge and integrate it with automatically generated models. We believe that in tandem with the development of modeling techniques it is also important to address these issues of usability and work flow integration.

Alongside the presented use of the data gathered, we also envisage it being beneficial in providing guidance to automated dynamic and static testing algorithms. Path and code region prioritisation is a challenge under most analysis approaches and it makes sense to prioritize code that is known to be subject to attacker influence. Furthermore, this data may allow one to begin analysis deeper in the code than necessarily starting from generic points of influence, such as file or socket access.

6. REFERENCES

[1] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. Aeg: Automatic exploit generation. In *Network and Distributed System Security Symposium*, pages 283–300, Feb. 2011.

[2] D. Blazakis. Differential reversing (or some better name). `http://dion.t-rexin.org/notes/2009/09/29/differential-reversing/`, 2009.

[3] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. Bap: a binary analysis platform. In *Proceedings of the 23rd international conference on Computer aided verification*, CAV'11, pages 463–469, Berlin, Heidelberg, 2011. Springer-Verlag.

[4] D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, pages 143–157, Washington, DC, USA, 2008. IEEE Computer Society.

[5] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 14th ACM conference on Computer and communications*

security, CCS '07, pages 317–329, New York, NY, USA, 2007. ACM.

[6] C. Cadar, D. Dunbar, and D. Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[7] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: Automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur.*, 12(2), 2008.

[8] V. Chipounov and G. Candea. Reverse engineering of binary device drivers with revnic. In *EuroSys*, pages 167–180, 2010.

[9] V. Chipounov, V. Kuznetsov, and G. Candea. The s2e platform: Design, implementation, and applications. *ACM Trans. Comput. Syst.*, 30(1):2, 2012.

[10] D. Duran, M. Miller, and D. Weston. Security defect metrics for targeted fuzzing. In *CanSecWest*, Vancouver, Canada, Mar. 2011.

[11] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[12] P. Godefroid, M. Y. Levin, and D. Molnar. Sage: Whitebox fuzzing for security testing. *Queue*, 10(1):20:20–20:27, Jan. 2012.

[13] S. Heelan. Automatic generation of control flow hijacking exploits for software vulnerabilities. Msc. dissertation, University of Oxford, September 2009.

[14] Hex-Rays. IDA. `http://www.hex-rays.com/products/ida/index.shtml`, 2005.

[15] M. G. Kang, S. McCamant, P. Poosankam, and D. Song. DTA++: Dynamic taint analysis with targeted control-flow propagation. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, San Diego, CA, Feb. 2011.

[16] J. Koret. MyNav, A Python plugin for IDA Pro. `http://joxeankoret.com/blog/2010/05/02/mynav-a-python-plugin-for-ida-pro/`, May 2010.

[17] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[18] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis. Path-exploration lifting: Hi-fi tests for lo-fi emulators. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, London, UK, Mar. 2012.

[19] D. Molnar, X. C. Li, and D. A. Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, pages 67–82, Berkeley, CA, USA, 2009. USENIX Association.

[20] N. Nethercote and J. Seward. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd international conference on Virtual execution environments*, VEE '07, pages 65–74, New York, NY, USA, 2007. ACM.

[21] A. Slowinska, T. Stancescu, and H. Bos. Howard: A dynamic excavator for reverse engineering data structures. In *NDSS*, 2011.

[22] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper.*, Hyderabad, India, Dec. 2008.

[23] Zynamics. BinNavi. `http://www.zynamics.com/binnavi.html`, 2010.

ThinAV: Truly Lightweight Mobile Cloud-based Anti-malware

Chris Jarabek
Department of Computer Science
University of Calgary
2500 University Drive NW
Calgary, AB, Canada T2N 1N4
cjjarabe@ucalgary.ca

David Barrera
School of Computer Science
Carleton University
1125 Colonel By Drive
Ottawa, ON, Canada K1S 5B6
dbarrera@ccsl.carleton.ca

John Aycock
Department of Computer Science
University of Calgary
2500 University Drive NW
Calgary, AB, Canada T2N 1N4
aycock@ucalgary.ca

ABSTRACT

This paper introduces ThinAV, an anti-malware system for Android that uses pre-existing web-based file scanning services for malware detection. The goal in developing ThinAV was to assess the feasibility of providing real-time anti-malware scanning over a wide area network where resource limitation is a factor. As a result, our research provides a necessary counterpoint to many of the big-budget, resource-intensive idealized solutions that have been suggested in the area of cloud-based security. The evaluation of ThinAV shows that it functions well over a wide area network, resulting in a system which is highly practical for providing anti-malware security on smartphones.

Keywords

Android, Malware, Cloud computing, Anti-virus

1. INTRODUCTION

The exponential rise in malware has caused countless research papers to begin with vacuous statements about the exponential rise in malware. Typically these are backed up by token citations to Gartner and Symantec reports, and occasionally a Department of Justice publication for good measure. We will omit this particular ritual.

Massive numbers of malware signatures and related updating issues have caused many anti-malware vendors to move parts of their product into the cloud over the last few years (e.g., [4, 18]). The answer to the question of how big anti-malware can get is therefore limitless. Few people seem to be asking the opposite question, however: how *small* can anti-malware be? This is an especially relevant question for mobile devices. We should note that by "small" we are naturally referring to a small amount of anti-malware software running on end hosts, but also a small amount of supporting infrastructure (ideally none). In other words, a tiny piece of software on the end host plus a massive local cloud infrastructure to maintain does not equal small.

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

We began answering this question for desktop computers.[1] We wrote a small Python program to intercept file accesses under Linux; we thus had our small desktop footprint, but somehow we had to check the files being accessed for malicious content, *without* creating or maintaining our own cloud infrastructure.

We observed that cloud-based anti-malware exists already, and it is freely-available in the sense that anyone can query it on the Internet. Specifically, we used Kaspersky, VirusChief, and VirusTotal.[2] All of these are similar insofar as a user of the service can upload any type of file, and receive a report about the malware (if any) that might be contained in the file. These WAN-based services acted as our anti-malware scanners; any access to a file that had not already been scanned by our system would be sent for scanning.

Our system was designed in a modular fashion, so it was able to leverage all these existing anti-malware services easily. Scanning requests were sent via the site's API if one existed (VirusTotal), otherwise they were made via HTTP requests and the results scraped from the HTTP responses. (We note that only VirusTotal had terms of service listed, which we abided by in addition to making attempts to minimize our traffic to all these services during testing and evaluation.)

On the desktop, we found that two factors conspired to create an underwhelming user experience. First, the multitude of different files being accessed resulted in poor local cache performance and many files being uploaded. Second, these files would often be accessed several at a time, and in rapid succession, aggravating latency issues. Furthermore, an orthogonal problem is that the files being uploaded could potentially contain sensitive data.

While things look grim on the desktop, the situation for mobile devices – notably Android – is such that our idea fits into the ecosystem better and works well. That is the topic of this paper.

A survey of malware encountered in the wild on Android, iOS and Symbian devices [9] found that all instances of malware for Android devices used application packages as their vector, meaning that users were unknowingly installing the malware on their device. This is not surprising, as Android applications can be installed from an arbitrarily large number of places, unlike the one-stop (and one-stop only!) shopping for iOS applications. There are multiple major An-

[1] We only summarize our desktop implementation and experiments here due to space constraints; full details are in [14].
[2] `kaspersky.com`; `viruschief.com`; `virustotal.com`.

droid app markets, even more minor markets, and apps can be downloaded and installed from the Internet or via USB. Furthermore, it is relatively trivial to construct and release a Trojanized version of a legitimate application.

Clearly there is a need for anti-malware protection on Android. In fact, Google has announced that due to the spate of malware on their market, they have developed their own internal anti-malware scanning system called Bouncer, which performs automated scanning of apps submitted to the market [16]. But this is just one application source of many, and this is where our system, ThinAV, fits in.

ThinAV provides lightweight cloud-based anti-malware protection for Android devices, combining a small Android client with the ability to leverage multiple anti-malware services on the Internet. Android effectively forces each application to run as a different user, thus limiting what we need to scan, as one app cannot modify another. We can reasonably constrain ThinAV to look at apps as they are installed, combined with a "killswitch" to manage cases of *post hoc* detection. This addresses file access latency, but also privacy: only apps, not data, need to be scanned, and ThinAV's design proxies scan requests so that individual users cannot be easily profiled by their IP addresses.

Section 3 presents ThinAV's architecture, after the related work in Section 2. Section 4 extensively evaluates ThinAV, and is followed by a discussion of ThinAV's limitations and our conclusions in Sections 5 and 6, respectively.

2. RELATED WORK

Cloud-based malware scanning as posited in [19, 20] was a significant source of inspiration for ThinAV. Their system, CloudAV, has end hosts run a lightweight client (300 LOC in Linux, and 1200 LOC in Windows) which tracks and suspends file access requests until a file has been scanned. This is the only lightweight part; CloudAV relies on a local cloud service consisting of twelve parallel VMs, ten of which run different anti-virus engines, and two run behavioral detection engines. These dedicated scanning servers run in a LAN environment, where the performance hit from network latency and system load is minimal. Even an extension of the CloudAV work to a mobile setting [21] failed to provide any information on how fast their solution operated in the lower-bandwidth / higher-latency mobile realm. ThinAV, by contrast, is truly lightweight. It has a small client that runs on the end host and it relies on already-existing anti-malware services on the Internet.

Many cloud-based anti-malware systems (e.g., [17, 5, 6, 4]) are an exercise in load balancing. Well-provisioned cloud servers perform intensive processing, in concert with clients that handle less demanding processing or operations that require client-side context. (In some cases, just extra processing power and not a cloud is needed: one system validates the contents of a mobile device when it is connected to a desktop or laptop computer via USB [7].) Some systems [13, 22, 2] tilt the balance and make the client a straightforward source of security data for the cloud to process, but again this needs resources on the server side if not the client side. ThinAV sidesteps this by combining a lightweight client with the ability to leverage existing services.

At the other extreme is anti-malware that is based on the end host, per the traditional anti-malware model. This presents a problem on resource-constrained mobile devices, and consequently these mobile systems tend to employ various generic detection heuristics, such as battery consumption [15], memory consumption [12], and heuristic (mostly permission-based) rules [8]. Running solely on a mobile device is neither lightweight nor does it allow use of existing services, though.

Finally, Meteor [1] draws on existing information sources such as developer registries, application databases and remote application killswitches to provide single-market security guarantees in a multi-market environment. While Meteor could potentially provide a framework in which ThinAV could operate, the server-side component is as yet unimplemented.

3. SYSTEM ARCHITECTURE

In this section we present the ThinAV anti-malware system. We begin with an overview, followed by a threat model, and then describe each ThinAV component.

3.1 Overview

ThinAV is an anti-malware system for Android which offloads the chore of scanning to existing third-party malware scanning services. ThinAV was designed to be lightweight, modular, extensible and has been tested with freely available online services such as Kaspersky, VirusChief, VirusTotal and ComDroid [3]. These scanning services all behave similarly in receiving cryptographic hashes of files or the binaries themselves as queries, and returning a scan result.

As shown in Figure 1, ThinAV consists of two main components: (1) an Android client, which submits applications for scanning and (2) a server which submits received files to third-party scanning services and notifies the client in the event of malware detection. The client software consists of modifications to the Android OS package manager as well as a client app for user notification of the scan result. A Killswitch module acts as a periodic post-installation scanner for installed apps. For performance reasons (described in Section 4), the server caches scan results in order to return them faster to clients.

Aside from performance and power consumption reduction, a clear benefit of splitting the client and server is the ability to update scanning modules without having to update the client code. This enables on-demand addition or removal of scanning services.

3.2 Threat Model

Android provides strong application isolation by assigning each application an unprivileged unique UNIX user id (UID). Once apps are installed, the underlying Linux kernel ensures isolation between apps and grants privileges according to the pre-approved permission request (displayed to the user at install-time). Under this security model, the key threat to the user and OS originates primarily from malicious apps that are installed voluntarily by a user, rather than from traditional vectors for malware such as drive-by-downloads and malicious executables [9]. Our threat model assumes:

1. Side-loaded[3] apps cannot be executed without being installed through the OS-provided `PackageInstaller`,

[3]Side-loaded apps are installed through mechanisms other than the official Google Play Store (e.g., third-party markets, file downloads).

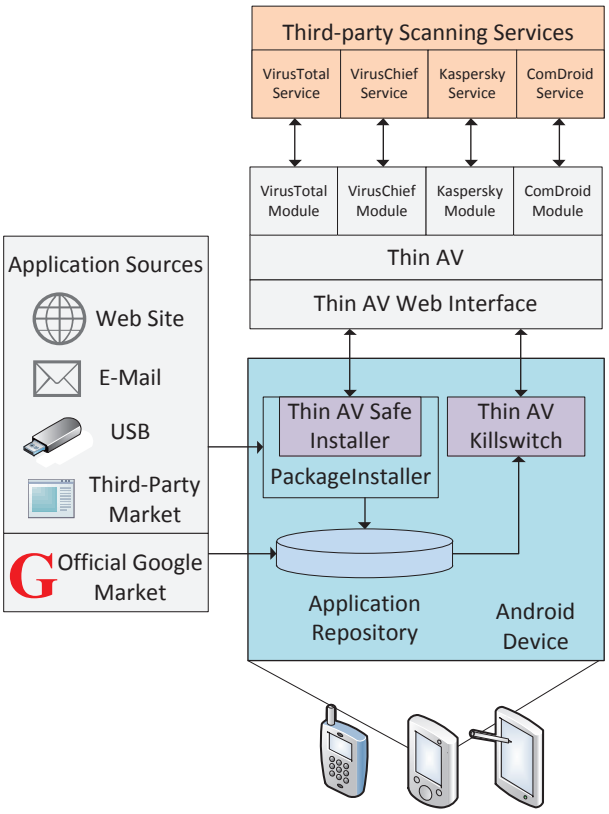


Figure 1: System architecture diagram for ThinAV.

a low-level framework responsible for completing the installation process.

2. Once installed, applications cannot modify their code dynamically. The only way to modify code or functionality is through an application update, which in itself is a new installation (that preserves user data) bound to Assumption 1.

3. The OS and pre-installed system applications are trusted.

3.3 Server

The ThinAV server component is responsible for receiving scan requests and submitting them to the scanning modules on behalf of the client. The ThinAV server is currently implemented in Python using the Flask[4] 0.8 web application micro-framework and runs on Linux. The server is designed in a modular object-oriented fashion, where a parent class provides scanning modules (i.e., subclasses) with all the functionality needed for searching and updating the local cache, as well as uploading binaries via HTTP `POST` requests.

To improve performance, the ThinAV server uses a local cache, which is implemented as a flat file containing previous scan results. While most online malware scanning services cache scan results, the latency in returning results (even those which are cached) was found to be unacceptable in our test conditions (see Section 4). Thus, whenever a scanning module is instantiated, the local cache is first checked.

[4]`http://flask.pocoo.org/`

The cache holds an MD5 hash of each scanned file, the full path to the file, the number of times ThinAV has been asked to analyze the file, the last time such an access has occurred, the infection status of the file, a note for additional scan details, and the module that was used when the file was analyzed.

3.4 Safe Installer

Safe Installer is the ThinAV component on the client responsible for preventing malicious applications from being installed. To build Safe Installer, we modified the Android Package Installer framework,[5] the system code in charge of parsing Android packages to verify integrity, and later completing the installation or update process by creating a new UID (if necessary) and placing files in the appropriate directories. All side-loaded applications must go through the Android Package Installer for installation, making this a ideal choke-point for placing our ThinAV client.

Specifically, we modified the `PackageInstallerActivity` class to make use of `ThinAvService`, a new service class which communicates with the Thin AV server described in the previous section. The service provides a single public function `checkAPK`, accessed via an interface defined using the Android Interface Definition Language. The `checkAPK` function takes the file system path of the Android app package (APK) being installed, reads the file and creates a cryptographic hash of the APK. This hash is then sent to the ThinAV web application, which returns a scan report, if such a report exists. If no scan report exists, the APK is uploaded to ThinAV where it is passed off to one of the third-party scanning services. When a scan result is returned, that result is passed back to `ThinAvService` and `checkAPK` then returns a Boolean value indicating whether or not the installation should be allowed to proceed. The `PackageInstallerActivity` then allows or prevents the installation of the application, displaying the appropriate information dialogs to the user, where necessary.

3.5 Killswitch

Safe Installer can prevent the installation of applications known to be malicious. However, Safe Installer will be unable to prevent the installation of malicious apps in two cases. First, when a malicious application was installed on a device prior to the installation of ThinAV; and second, when an application was installed on a device but was not flagged as malicious at the time of installation. A Killswitch was developed to address these two scenarios. The Killswitch operates independently of any specific application installation mechanism, making it ideal for the multi-market ecosystem available on Android devices.

The Killswitch was developed as a standalone Android application capable of communicating with the ThinAV server, similar to the Safe Installer, but invoked on-demand rather than automatically at install-time.

The Killswitch has three different functions available to the user. (1) It can upload all applications to ThinAV for analysis (if those applications are not in the ThinAV local cache); (2) it can manually check if any non-system applications on the device have been flagged as malicious; and (3) it can regularly check the device for malicious applications

[5]Specifically, we modified the Android 2.3.7 source code which was in use by most deployed Android devices [23].

using a scheduled event. In the current implementation the killswitch is scheduled to run every 15 minutes.

When the Killswitch is checking for malicious apps, it uses the `PackageManager` class to locate all Android packages installed on the device. For each package, the Killswitch reads the meta-data, creates a hash of each app's byte contents, and a collection of all package hashes is sent to the ThinAV server. If a package has already been hashed by the Killswitch, then the hash is stored in a file which is only accessible to the Killswitch. This hash can then be retrieved much more quickly than recomputing the hash every time the device is fingerprinted. If any of the hashes sent to the ThinAV server are found to be from a malicious app, the user is notified of the infection, and presented a list of applications suspected to be malicious. The user can then choose to initiate the removal of those applications.

3.6 Scanning Modules

ThinAV can be configured to offload scanning to any third-party malware scanning service with a public API or web interface. The system currently has modules for four scanning services that are freely available online: Kaspersky, VirusChief, VirusTotal, and ComDroid. These services all behave similarly insofar as a user can upload any type of file (executable, data, etc.) through the website of the service and receive a report as to any malware that might be contained in that file. Unfortunately, these scanning services are based on proprietary anti-malware engines, and as such, the exact details of the engines underpinning these services are very closely held trade secrets. Therefore, the exact capabilities and limitations of these services with respect to threat detection are not publicly known.

The currently implemented modules and their descriptions:

- Kaspersky – Offers a free service that uses a proprietary anti-malware engine for scanning files that are 1MB or smaller in size.
- VirusChief – A multi-engine anti-malware scanning service with a 10MB file size limit.
- VirusTotal – Scans uploaded files up to 20MB in size with 42 different scanning engines. VirusTotal has a public API to interact with its services.
- ComDroid - While not an anti-malware engine, ComDroid can identify potential vulnerabilities in Android apps by performing static code analysis. The ComDroid module identifies scanned applications as being "at risk" as opposed to being "infected".

The ThinAV server is currently configured to select scanning services based on the average amount of time each module takes to scan a file. We found the fastest service to be Kaspersky, followed by VirusChief, ComDroid, and VirusTotal. If any scanning module returns an error from an attempted online scan, then the next module in the priority sequence is selected. If all four scanning modules fail, a general error code is returned to the device that originated the request. Performance measurements for the scanning modules are discussed in the following section.

4. THINAV EVALUATION

This section presents the results of our evaluation of ThinAV. All development and testing was done on the Android emulator provided in the SDK. The emulator enabled rapid development on different versions of the Android operating system, and allowed for changes to be made to the Android source code.

Working on the emulator presents evaluation issues with respect to network performance. Because ThinAV is heavily reliant on the network, link speed has a direct impact on performance. On a mobile device like a cell phone, the speed of the cellular connection can vary based on the location of the user, radio interference, the load on the cellular network, as well as other factors. Due to the challenges involved in cellular network measurements, we use the results of Gass et al. [10].

Number of Apps	1022
Mean App Size	2.65 MB
Median App Size	1.78 MB
Minimum App Size	0.02 MB
Maximum App Size	37.06 MB
Proportion of Apps <1 MB	34.64 %
Proportion of Apps <10 MB	97.16 %
Proportion of Apps <20 MB	99.51 %

Table 1: General file size characteristics of the Android test data set.

4.1 Data Set

The evaluation process was performed with a collection of apps downloaded from the official Google Play Store (known as the Android Market at the time of data collection) using a custom crawler. We downloaded the top 50 free apps (as ranked by user ratings) in each application category on January 3, 2012. The majority of package downloads were successful, with 28 downloads causing repeated failures. This resulted in 1,022 apps spread across 21 application categories, with each category having between 46 and 50 packages. Table 1 summarizes the key file size statistics of the data set.

4.2 Malware Detection

We first uploaded the entire data set to the VirusTotal scanning service to to confirm that VirusTotal, and therefore other scanning services are capable of correctly receiving and scanning Android applications. This initial scan also allowed us to obtain a baseline of detection. That is, to see how many apps in our initial data set contain malware.

VirusTotal flagged several possible instances of malware in the data set downloaded from the Google Play Store. Of the 1,022 apps uploaded, 1,019 were scanned (three apps were skipped due to size restrictions) and 27 were flagged as malware by at least one scanning engine. One package was flagged as malware by four different engines, nine packages were flagged by two engines, and the remaining seventeen packages were flagged as malware by a single engine. Table 2 provides details on some of the commonly flagged samples. The most commonly identified sample was from the `Adware.Airpush` family. However, the majority of these samples were identified by a single scanning engine (DrWeb), which raises the possibility of this being a false positive. The next most common sample was `Plankton`, which was identified by a variety of scanning engines. The remaining

Sample Name	Malware Type	Count	Detection Engine(s)
`Adware.Airpush(2, 3)`	Adware	15	DrWeb, Kaspersky
`Plankton (A, D, G)`	Trojan	6	Kaspersky, Comodo, NOD32, Trend Micro
`SmsSend (151, 261)`	Dialer	2	DrWeb
`Rootcager`	Trojan	2	Symantec

Table 2: Most frequent samples of malware detected in Google Market data set. Detection engine refers to which VirusTotal scanning engines detected the sample.

malware samples had far fewer occurrences in the data set.

While the test data set only suggested that 6 of the AV engines used by VirusTotal are capable of detecting Android malware, we later confirmed that as many as 26 (more than half of the VirusTotal scanning engines) are capable of detecting some form of Android specific malware.

4.3 AV Scanning Module Performance

We developed a testing program which uploaded files of different sizes to each of the scanning services at specific time intervals. This program was designed to measure the response time of each service. The program submitted 12 files (of sizes 0 KB, 1 KB, 2 KB, 4 KB, 8 KB, 16 KB, 32 KB, 64 KB, 128 KB, 256 KB, 512 KB and 1023 KB) for scanning in random order over an 8 day window. The files were created by a script which produces files of a specified size filled with pseudo-random bits with the expectation that files generated in such a way would have an extremely low probability of being flagged as malware by one of the scanning services, or exist in the service cache. Test files were uploaded in a pseudo-random order every time the test program was run to overcome any penalty that might be incurred against the first file being uploaded due to DNS lookups.

Results.

For each of the three scanning services, several hundred response time measurements were recorded. A cursory review of the data showed a handful of extreme outliers for each service. Any measurement beyond two standard deviations of the mean was classified as an outlier. This threshold was chosen because it eliminated the most extreme results, while retaining the vast majority of the data.

A comparison of the measurements from the three services shows a clear difference of nearly an order of magnitude between the performance of VirusTotal and the other two scanning services. The average response times from Kaspersky and VirusChief range from 1.54 – 14.49 seconds and 6.82 – 28.70 seconds respectively, while the response times from VirusTotal range from 1.21 – 229.28 seconds (though the latter range becomes 148.92 – 229.28 seconds, when only non-zero file sizes are considered). The upload portion of VirusTotal shows response times similar to Kaspersky with response times ranging from 1.94 – 11.74 seconds.

With the outliers removed, the response time data was plotted (see Figures 2, 3, and 4) in an attempt to determine the correlation between file size and service response time for the three scanning services. Each of the figures shows the upload file size plotted versus the response time for each of the three scanning services, and Figure 5 graphs the upload and response speed of VirusTotal. Kaspersky and VirusChief both show a similar positive correlation between file size and response time, with the VirusChief data being positively shifted on the y-axis (and therefore slower) by roughly fifteen seconds. VirusTotal, on the other hand, shows little if any relationship between file size and response time. The trend of the VirusTotal response time data is slightly negative and has a much larger y-intercept than either of the other two scanning services. Conversely, the upload portion of the VirusTotal scan shows a trend very similar to Kaspersky. With this data, we produced a set of linear equations which approximate the performance of the scanning services as a function of the number of bytes in the file (see Table 3).

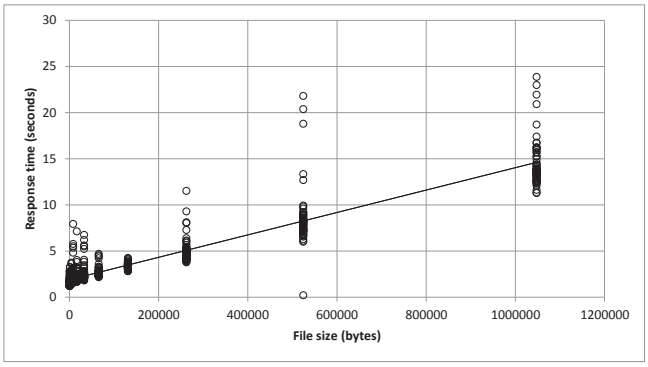


Figure 2: Scan response time versus file upload size for the Kaspersky virus scanner service.

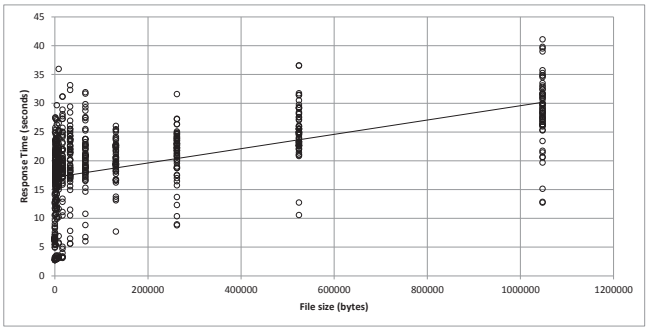


Figure 3: Scan response time versus file upload size for the VirusChief virus scanner service.

Discussion.

It is not surprising that Kaspersky, which only scans with a single anti-virus engine, returns the fastest results, and VirusChief, which scans with six anti-virus engines is roughly fifteen seconds slower than Kaspersky when scanning a similarly sized file. We attribute the erratic VirusTotal performance to the company's prioritization of scanning requests. VirusTotal assigns the lowest priority to requests that are sent via their formal API, and the response appears to not be dependent on the size of the uploaded file, but rather on how busy the VirusTotal scanning service is at any given

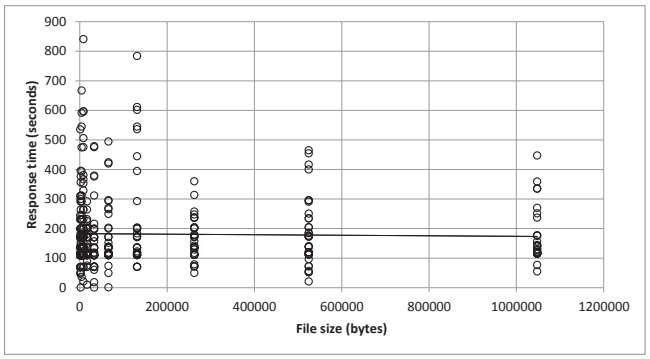


Figure 4: Scan response time versus file upload size for the VirusTotal virus scanner service.

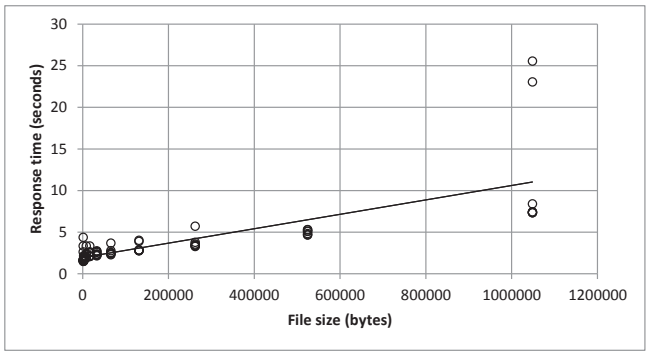


Figure 5: Upload response time versus file upload size for the VirusTotal virus scanner service when uploading a file and not polling for a scan result.

moment. This distinction is made much clearer when comparing the time required to scan a file with VirusTotal with the time required to merely upload a file to VirusTotal.

4.4 ComDroid Performance

To evaluate ComDroid, we uploaded each of the 1,022 apps in our data set to the ComDroid scanning service over a two day period. For each upload, we recorded the time required for a scan report to be returned.

Results.

Of the 1,022 packages uploaded to ComDroid, 993 were scanned, with the remainder being rejected by the server due to a 10 MB size limitation. Of the 993 packages scanned by ComDroid, 8 returned a scan error, resulting in 985 valid scan results. The mean response time was 40.67 seconds (σ = 77.60 seconds), and the median response time was 18.63 seconds. Figure 6 shows the response time plotted as a function of the package size, and the exact function is specified in Table 4. There is a clear positive linear relationship between package size and scan time, although numerous outliers are present.

Kaspersky	$f(x) = 10^{-5} \times x + 1.891$
VirusChief	$f(x) = 10^{-5} \times x + 17.133$
VirusTotal	$f(x) = -9 \times 10^{-6} \times x + 182.98$
VirusTotal (Uploads)	$f(x) = 9^{-6} \times x + 1.947$

Table 3: Linear equations for each of the three scanning services derived from Figures 2, 3, 4, and 5. Equations calculate the response time for each scanning service for a file x bytes in size.

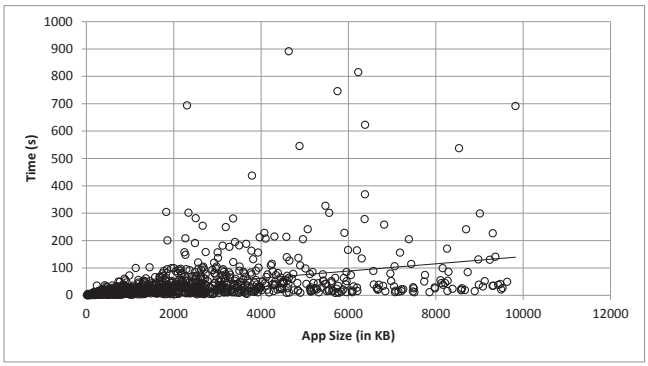


Figure 6: Response time of the ComDroid service as a function of package size.

ComDroid	$f(x) = 0.0132 \times x + 9.6893$

Table 4: Linear equation for the ComDroid scanning service.

Discussion.

The performance of the ComDroid service is somewhat similar to both the Kaspersky and VirusChief services (Section 4.3). However, the linear trend is much less prominent. The most likely explanation for this lies in the nature of the analysis performed by ComDroid. ComDroid is a static code analysis tool, and as such, it is safe to assume that the time required to analyze an Android app is more directly influenced by the amount of code in the package, than the total size of the package. Given that many apps contain numerous resource files (images, sounds, video, etc.) which are not scanned by ComDroid, it is easy to imagine how a package might have a large size, but a relatively small amount of code. It is quite likely that the observed linear trend is much more a result of the upload time and not the code-to-resource-file ratio of the package.

ComDroid detects a wide variety of "programming errors". In our experiment, 971 of 985 apps (98.6%) were flagged by ComDroid as containing some type of code that is vulnerable to exposed communication. This suggests that, at least in its current form, simply flagging an application as being "at risk" if there is any instance of exposed communication would effectively cripple the ability of users to install apps on their device. Chin et al. [3] (the ComDroid authors) suggest that in their data, after manual inspection of warnings, only about 10–15% were genuine vulnerabilities.

We believe that there is a place for ComDroid (and other tools like it) in the ThinAV architecture. However, the behavior of this ThinAV module would likely have to be adjusted over time to prevent excessive false positives. This could be done by creating thresholds which would flag a package as vulnerable if it had significantly more exposed surfaces than average for a given type of warning.

Network Configuration	Upload Speed (KBps)	Download Speed (KBps)
Typical 3G	16.25	84.13
Ideal 3G	1792.00	1792.00
Typical WiFi	190.38	155.38
Ideal WiFi	76800.00	76800.00

Table 5: Network speeds used for evaluating the mobile implementation of ThinAV.

Network Configuration	Small File	Medium File	Large File
Ideal 3G	0.034 s	0.232 s	0.293 s
Typical 3G	0.041 s	0.239 s	0.300 s
Ideal WiFi	0.034 s	0.231 s	0.293 s
Typical WiFi	0.035 s	0.233 s	0.294 s

Table 6: Time required to check an package in ThinAV, for three different file sizes, and four different network configurations, assuming the scan result is already cached by the ThinAV server

4.5 Safe Installer Performance

The performance of the Safe Installer is based on three factors: the size of the package being scanned, the speed of the network to which the device is connected, and whether or not the package being installed has already been scanned by ThinAV. To evaluate the Safe Installer, we use three different file sizes: 0.76 MB (small), 1.78 MB (medium), and 3.56 MB (large), corresponding to the median size of apps in the category with the smallest median size (medical apps), the median size for the entire data set, and the median size of apps in the category with the largest median size (educational apps). We also use the "ideal" and "typical" 3G and WiFi speeds from prior work [10, 11] (see Table 5).

Results.

The best case scenario for the performance of the safe installer is when the package being installed has already been scanned by ThinAV. In this case, the cost for performing an install time check is equal to the time required to hash the installing application, send the hash to ThinAV, look up the scan result, and return the scan result.

We measured the time required to hash a small, medium, and large application on the Android emulator, and took the average of five runs for each size. The emulator was able to calculate hashes of small files in 0.033 seconds, medium files in 0.231 seconds, and large files in 0.293 seconds. We recorded the amount of data uploaded and downloaded during transmission of the hash to the ThinAV server, as well as the reception of the result. This was approximately 200 bytes (100 up, 100 down), although this amount varied slightly with the file being scanned. Finally, the cost of the ThinAV server performing a cache lookup was 0.0002 seconds. Table 6 summarizes the results for this best case scenario. In general, even the largest file over the slowest network only takes 0.3 seconds to check with ThinAV.

The worst case scenario is when the application being installed has not been scanned by ThinAV, and therefore the whole package must be uploaded to ThinAV, which must then upload the package to one or more of the third-party scanning services. Using the formulæ in Tables 3 and 4, and the file sizes and network speeds above, it is possible to compute the time required to upload and scan these files at install time. We note that when calculating the time required to scan a package, we add both the time to scan the package with an anti-virus module as well as the time for scanning with ComDroid.

Network Configuration	Small File	Medium File	Large File
Ideal 3G	36.56 s	98.13 s	170.29 s
Typical 3G	84.66 s	210.00 s	394.14 s
Ideal WiFi	36.13 s	97.14 s	168.31 s
Typical WiFi	40.23 s	106.68 s	187.39 s

Table 7: Time required to check an package in ThinAV, for three different file sizes, and four different network configurations, assuming the scan result is not cached by ThinAV.

Table 7 summarizes the results for this worst case scenario. In general, the time required to upload and scan an Android package ranges between 36 seconds and 394 seconds, depending on the size of the file and the speed of the network.

Discussion.

The best case scenario, where ThinAV already has a cached scan result, is extremely fast. At 0.3 seconds, this check would be unnoticeable to a user. On the other hand, if the file must be uploaded and scanned, this process could take as long as seven minutes. This could be seen as a serious inconvenience to the user, but considering that this check would only take place when a user is installing an app that has never been seen by the ThinAV server, large-scale deployment should make this an infrequent occurrence. Additionally, given that ThinAV could be primed with packages from a variety of sources, including regular downloads of applications from various application markets, upload of applications by developers, and the upload of applications by other users running ThinAV, the chance that a user would have to upload a package for scanning at install time could be made very rare. Of course, only the evaluation of a wide-scale deployment can provide confirmation of our intuition.

4.6 Killswitch Performance

During normal operation, we expect the most frequently used functionality of ThinAV to be the killswitch service which is periodically activated and checks for revoked apps. To evaluate the performance of the killswitch, we examine several factors: (1) the cost of hashing apps to generate a system fingerprint; (2) the network cost associated with uploading the fingerprint; (3) the cost of looking up the hashes on the ThinAV server; and (4) the network cost associated with returning those hashes to the client. We also consider a manual upload feature, in which all packages currently installed are submitted for scanning. This is required to scan applications that were installed prior to enabling ThinAV.

In general, the time required for the killswitch to perform a check for revoked apps without uploading any full packages to the ThinAV server will be:

$$
\begin{aligned}
h_1 &= \text{time to hash all packages} \\
s_u &= \text{hash upload size} \\
sp_u &= \text{link upload speed} \\
c_1 &= \text{cache lookup time} \\
s_d &= \text{response download size} \\
sp_d &= \text{link download speed} \\
t_1 &= h_1 + \frac{s_u}{sp_u} + c_1 + \frac{s_d}{sp_d}
\end{aligned}
\tag{1}
$$

Because the cost of performing a manual upload of missing packages is dominated by upload and scanning costs (similar to the safe installer above), we include only these costs in the calculation. The time required for the killswitch to manually upload missing packages is:

$$
\begin{aligned}
s_u &= \text{package upload size} \\
sp_u &= \text{link upload speed} \\
t_s &= \text{time to scan all applications} \\
t_2 &= \frac{s_u}{sp_u} + t_s
\end{aligned}
\tag{2}
$$

To test the performance of the hashing function, we installed the top five apps from each of the 21 Google Play app categories (on top of the 5 default non-system apps) on the Android emulator. After installation, we generated a complete system fingerprint (i.e., a hash for each app installed on the user partition) ten times and recorded the average time. This represents the worst case scenario in which none of the apps on the device have been hashed before, and all hashes must be computed. Next, we generate another ten fingerprints and record the average time. The cache created in the previous test was left intact, however. This represents the best case scenario in which all of the apps on the phone have already been hashed and the phone fingerprint is stored locally.

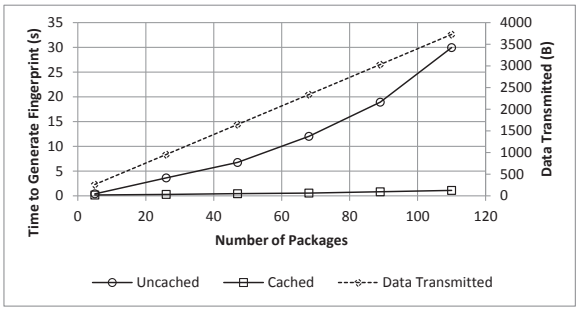
Under normal use it is likely to expect that the typical scenario would in fact be the best case scenario, or very close to it. After the first fingerprint has been generated, the only time an app will have to be hashed is when it has not been seen by the killswitch, meaning it has just been installed. Unless a user installs numerous apps between the scheduled runs of the killswitch, it is likely the number of apps that need to be hashed would be near zero.

Combining the hashing performance with the file size data for the data set, the scanner performance functions in Tables 3 and 4, and the experimental network performance measurements from [10], we calculate the cost of performing manual uploads, as well as the cost of fingerprinting based on Equations 1 and 2.

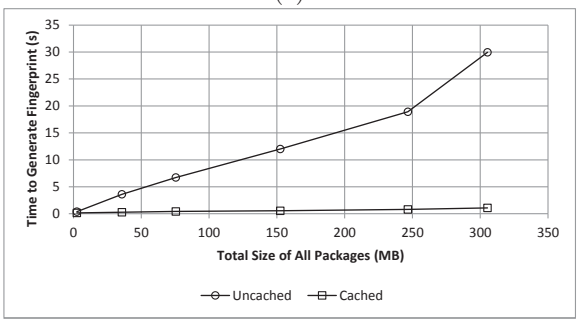
Results.

Figure 7 shows the best (cached) and worst (uncached) case scenarios for the fingerprint generation time as a function of both the number of packages on the device and the total size of those packages.

It is clear that time to generate a system fingerprint grows linearly with both the number and size of packages on the device. In the worst case, with 110 apps on the device, it only takes 29.95 seconds to generate a system fingerprint. The best case scenario is better, with a fingerprint being generated in 1.09 seconds for the same 110 apps when the fingerprint has been cached.



(a)



(b)

Figure 7: Time required to generate a complete system fingerprint as a function of the number of packages installed on the device (a) and the total size of those packages (b). Both figures show the average time when all of the package hashes have been stored (cached) and when none of the package hashes are stored (uncached). Figure (a) also includes the number of bytes sent and received when communicating the fingerprint to the ThinAV server.

Data usage grows linearly with the number of packages on the device. The data consumption ranges from 3.64 KB for 110 apps, down to 261 bytes for 5 apps. The majority of this transmission is in the form of the uploaded fingerprint, as the response from ThinAV only downloads 70 bytes from the server when the fingerprint contains no hashes corresponding to malicious apps.

The current implementation of the ThinAV client is scheduled to run the killswitch service every 15 minutes. Table 8 shows how much data would be consumed by ThinAV (under the current configuration) over different lengths of time

Interval	Data Consumption (5 Apps)	Data Consumption (110 Apps)
1 Day	24.47 KB	349.41 KB
1 Week	171.28 KB	2.39 MB
1 Month	5.19 MB	74.04 MB

Table 8: Data consumption of ThinAV killswitch over different time periods, for 5 and 110 apps installed on the device, assuming the killswitch is scheduled to run every 15 minutes.

Scenario	Time (seconds)
110 apps / ideal 3G / no hashes cached	26.206
110 apps / typical 3G / no hashes cached	26.430
110 apps / ideal WiFi / no hashes cached	26.204
110 apps / typical WiFi / no hashes cached	26.223
110 apps / ideal 3G / all hashes cached	3.424
110 apps / typical 3G / all hashes cached	3.478
110 apps / ideal WiFi / all hashes cached	3.423
110 apps / typical WiFi / all hashes cached	3.428
26 apps / ideal 3G / no hashes cached	1.034
26 apps / typical 3G / no hashes cached	1.258
26 apps / ideal WiFi / no hashes cached	1.032
26 apps / typical WiFi / no hashes cached	1.051
26 apps / ideal 3G / all hashes cached	0.285
26 apps / typical 3G / all hashes cached	0.339
26 apps / ideal WiFi / all hashes cached	0.285
26 apps / typical WiFi / all hashes cached	0.290

Table 9: Time required to complete the fingerprinting operation for different numbers of applications, network performance, and caching scenarios.

and with two sets of installed apps (5 and 110).

Using the same network measurements from Section 4.5, the measured fingerprint generation times, and data transmission totals, it is possible to compute a variety of potential running times for the entire fingerprinting operation of ThinAV killswitch using Equation 1. These values are summarized in Table 9.

	Total Data Uploaded (MB)		
Scenario	10 Apps	25 Apps	50 Apps
Small Apps	7.643	19.108	38.216
Medium Apps	17.775	44.438	88.875
Large Apps	35.570	88.925	177.850

Table 10: Total upload sizes used for calculations of manual scanning performance.

For calculating the cost of manually uploading missing packages, we use the package size averages from Section 4.5 and use three sets of apps (10, 25, and 50 apps). Table 10 summarizes the total amount of data that would be uploaded for different numbers of apps of different sizes. The upload times for the different numbers and sizes of apps are summarized in Table 11. Using the size and quantity of each app, the scanning time could then be computed using the equations in Tables 3 and 4. These results are summarized in Table 12. Finally, referring to Equation 2, it is possible to compute the time required to upload and scan missing apps under different scenarios.

The best case scenario is when ten small apps are uploaded and scanned over an ideal WiFi connection. This takes 289.2 seconds, or just under five minutes. The worst case scenario is where 50 large apps are uploaded and scanned over a typical 3G connection. This operation takes 17351.2 seconds, or nearly five hours. However, if the same operation is performed over a typical WiFi connection, the time required to complete this one-time operation drops by more than half, to 1.95 hours.

		Upload Time (Seconds)		
Scenario		10 Apps	25 Apps	50 Apps
Ideal 3G	Small Apps	4.367	10.919	21.837
	Medium Apps	10.157	25.393	50.786
	Large Apps	20.326	50.814	101.629
Typical 3G	Small Apps	485.360	1213.400	2426.801
	Medium Apps	1128.781	2821.953	5643.907
	Large Apps	2258.833	5647.082	11294.164
Ideal WiFi	Small Apps	0.102	0.255	0.510
	Medium Apps	0.237	0.593	1.185
	Large Apps	0.474	1.186	2.371
Typical WiFi	Small Apps	41.111	102.777	205.553
	Medium Apps	95.609	239.023	478.046
	Large Apps	191.326	478.315	956.630

Table 11: Upload times for the values in Table 10, for four different network configurations.

	Scanning Time (Seconds)		
Scenario	10 Apps	25 Apps	50 Apps
Small Apps / Kaspersky	161.021	402.552	805.104
Medium Apps / VirusChief	634.032	1585.080	3170.159
Large Apps / VirusChief	1104.893	2762.232	5524.465
Small Apps / ComDroid	107.224	252.563	494.796
Medium Apps / ComDroid	120.919	266.259	508.491
Large Apps / ComDroid	144.972	290.312	532.544

Table 12: Scan times for different numbers of apps with small, medium and large sizes, using conventional scanning engines (Kaspersky and VirusChief) as well as the Android-specific scanner, ComDroid.

Discussion.

During long-term use of ThinAV, fingerprinting installed apps is the only operation that would likely take place frequently. In the best case, the killswitch requires about 1 second of computation followed by less than 4 KB of data transmission for a set of 110 installed apps. This operation would be unnoticeable to a user, especially on a physical Android device, which is typically more powerful than the Android emulator.

In terms of data consumption, the 74 MB per month for uploading the fingerprint of 110 apps is non-trivial, particularly for users with pay-as-you-go type data plans. We note, however, that data consumption can be lowered by reducing the frequency with which the killswitch is run. Compressing the data for fingerprint submission and response retrieval is possible and would likely also reduce data consumption.

5. LIMITATIONS

We have shown that despite being in an early prototype form, ThinAV could be realistically deployed on actual devices as a free and lightweight anti-malware system. We now discuss some of the limitations of the prototype and the system overall.

OS modification. Because the Package Installer is part of the core Android OS, ThinAV cannot be installed on any Android device as an application. Instead, the underlying OS must be replaced with a ThinAV-enabled version. Alternatively, Google and other phone manufacturers could incorporate ThinAV directly into their own builds.

Test environment. All tests in our experiment were performed on the Android emulator. While we believe the em-

ulator provides accurate technical feasibility metrics, it also provides a lower bound on speed measurements. Physical devices are generally more powerful, but also have battery consumption concerns. Future work will evaluate battery consumption by ThinAV.

Third-party scanning services. ThinAV relies on the continual existence of third-party scanning services in a production capacity. The terms of service of these services may change, and services may also cease to exist. The decision on whether or not to support HTTPS connections is also out of ThinAV's control, as are denial-of-service attacks (which are possible for any cloud-based anti-malware). Fortunately, the modular design of ThinAV should help transparently replace scanning modules without updating clients.

6. CONCLUSIONS

Keeping malware in check for Android is a difficult problem; Android has a chaotic multi-market app environment and the ability for users to side-load apps of unknown provenance. We address this problem not by imposing a massive anti-malware regime, but by just the opposite. Our ThinAV system combines a lightweight footprint on an Android device, consisting of a safe installer and killswitch, with the ability to leverage multiple free, already-existing anti-malware services on the Internet. As only apps are scanned and requests are proxied through a ThinAV server, no personal or IP address data is leaked to outside services. Our experiments with performance and data consumption have shown that small is practical, especially if ThinAV is fully integrated into the Android app ecosystem and is already primed with scan results for popular apps.

Acknowledgment. This work has been supported in part by the Natural Sciences and Engineering Council of Canada via ISSNet, the Internetworked Systems Security Network.

7. REFERENCES

[1] D. Barrera, W. Enck, and P. C. van Oorschot. Meteor: Seeding a Security-Enhancing Infrastructure for Multi-market Application Ecosystems. In *IEEE Mobile Security Technologies*, 2012.

[2] J. Cheng, S. H. Wong, H. Yang, and S. Lu. Smartsiren: virus detection and alert for smartphones. In *5th International Conference on Mobile Systems, Applications and Services*, pages 258–271, 2007.

[3] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *9th International Conference on Mobile Systems, Applications, and Services*, pages 239–252, 2011.

[4] M. Chiriac. Tales from cloud nine. In *19th Virus Bulletin International Conference*, pages 83–88, 2009.

[5] B.-G. Chun and P. Maniatis. Augmented smartphone applications through clone cloud execution. In *12th Workshop on Hot Topics in Operating Systems*, 2009.

[6] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: making smartphones last longer with code offload. In *8th International Conference on Mobile Systems, Applications, and Services*, pages 49–62, 2010.

[7] B. Dixon and S. Mishra. On rootkit and malware detection in smartphones. In *2010 International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 162–163, 2010.

[8] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *16th ACM Conference on Computer and Communications Security*, pages 235–245, 2009.

[9] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In *1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 3–14, 2011.

[10] R. Gass and C. Diot. An experimental performance comparison of 3G and Wi-Fi. In *Passive and Active Measurement*, volume 6032 of *LNCS*, pages 71–80, 2010.

[11] IEEE Computer Society. Wireless LAN medium access control (MAC) and physical layer specifications enhancements for higher throughput, Oct. 2009. IEEE Std 802.11n-2009.

[12] M. Jakobsson and K.-A. Johansson. Retroactive detection of malware with applications to mobile platforms. In *USENIX HotSec*, 2010.

[13] M. Jakobsson and A. Juels. Server-side detection of malware infection. In *2009 New Security Paradigms Workshop*, pages 11–22, 2009.

[14] C. Jarabek. Towards cloud-based anti-malware protection for desktop and mobile platforms. Master's thesis, University of Calgary, 2012.

[15] L. Liu, G. Yan, X. Zhang, and S. Chen. VirusMeter: Preventing your cellphone from spies. In *Recent Advances in Intrusion Detection*, volume 5758 of *LNCS*, pages 244–264, 2009.

[16] H. Lockheimer. Android and security. `http://googlemobile.blogspot.com/2012/02/android-and-security.html`, Feb. 2012.

[17] L. Martignoni, R. Paleari, and D. Bruschi. A framework for behavior-based malware analysis in the cloud. In *Information Systems Security*, volume 5905 of *LNCS*, pages 178–192, 2009.

[18] C. Nachenberg, Z. Ramzan, and V. Seshadri. Reputation: A new chapter in malware protection. In *19th Virus Bulletin International Conference*, pages 185–191, 2009.

[19] J. Oberheide, E. Cooke, and F. Jahanian. Rethinking antivirus: executable analysis in the network cloud. In *USENIX HotSec*, 2007.

[20] J. Oberheide, E. Cooke, and F. Jahanian. CloudAV: N-version antivirus in the network cloud. In *17th USENIX Security Symposium*, pages 91–106, 2008.

[21] J. Oberheide, K. Veeraraghavan, E. Cooke, J. Flinn, and F. Jahanian. Virtualized in-cloud security services for mobile devices. In *1st Workshop on Virtualization in Mobile Computing*, pages 31–35, 2008.

[22] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos. Paranoid Android: versatile protection for smartphones. In *26th Annual Computer Security Applications Conference*, pages 347–356, 2010.

[23] D. Rowinski. More than 50% of Android devices still running Froyo. ReadWrite Mobile, 6 September 2011. `http://www.readwriteweb.com/mobile/2011/09/more-than-50-of-android-device.php`.

Abusing Cloud-Based Browsers for Fun and Profit

Vasant Tendulkar
NC State University
tendulkar@ncsu.edu

Joe Pletcher
University of Oregon
pletcher@cs.uoregon.edu

Ashwin Shashidharan
NC State University
ashdharan@ncsu.edu

Ryan Snyder
University of Oregon
rss@cs.uoregon.edu

Kevin Butler
University of Oregon
butler@cs.uoregon.edu

William Enck
NC State University
enck@cs.ncsu.edu

ABSTRACT

Cloud services have become a cheap and popular means of computing. They allow users to synchronize data between devices and relieve low-powered devices from heavy computations. In response to the surge of smartphones and mobile devices, several cloud-based Web browsers have become commercially available. These "cloud browsers" assemble and render Web pages within the cloud, executing JavaScript code for the mobile client. This paper explores how the computational abilities of cloud browsers may be exploited through a Browser MapReduce (BMR) architecture for executing large, parallel tasks. We explore the computation and memory limits of four cloud browsers, and demonstrate the viability of BMR by implementing a client based on a reverse engineering of the Puffin cloud browser. We implement and test three canonical MapReduce applications (word count, distributed grep, and distributed sort). While we perform experiments on relatively small amounts of data (100 MB) for ethical considerations, our results strongly suggest that current cloud browsers are a viable source of arbitrary free computing at large scale.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed Applications; H.3.4 [**Systems and Software**]: Distributed Systems

General Terms

Security

Keywords

Cloud computing, web browsers, access control

1. INTRODUCTION

Software and computation is increasingly moving into "the cloud." Infrastructure-as-a-Service (IaaS) and Platform-as-a-Service (PaaS) have effectively commoditized computing resources, enabling pay-per-use computation. For example, in April 2012, Amazon's on-demand instances of EC2 cost as little as US$0.08 per hour [4]. This shift towards cloud computing provides many benefits to enterprises and developers. It consolidates hardware and maintenance, and it allows organizations to purchase only as much computing as they need. Equally important, the ubiquity of cloud providers and sophisticated interfaces make incorporating cloud functionality simple for virtually any piece of software.

Cloud computing has substantially benefited smartphones and mobile devices, relieving them of computation, storage, and energy constraints. Recently, several commercial ventures have deployed infrastructures for rendering Web pages in the cloud (e.g., Amazon Silk [5], Opera Mini [23], and Puffin [12]). The obvious benefit to this architecture is relieving the mobile device from the graphical rendering. However, this is less of a concern for newer, more powerful smartphones. Such devices benefit more from the cloud server downloading the many parts of a Web page using high-bandwidth links and only using the higher-latency, last-mile wireless network once. Proxy-based Web page rendering has existed in literature for more than a decade [17, 16, 19, 8, 9] and is of continued interest [10, 30]; however, it was not until the recent surge in smartphone popularity that commercial offerings became more widespread and well provisioned.

Cloud-based Web browsers (which we call "cloud browsers" for short) are often provisioned to exceed the computational power and functionality of a desktop browser. For example, Cloud Browse runs a modified Firefox desktop browser [3]. Over the past decade, websites have evolved into full-fledged applications executing nontrivial computations written in JavaScript. Cloud browsers must execute this JavaScript. Given this mix of powerful cloud-based computing ability and a substrate for general executions, we sought to investigate whether opportunities for exploiting unintended functionality were now possible. Specifically, was it now possible to *perform arbitrary general-purpose computation within cloud-based browsers,* at no cost to the user?[1] A successful outcome would demonstrate the ability to perform *parasitic computing* [7] within the cloud browser environment, whereby it is transformed into an unwitting computational resource merely through supplying browser requests.

In this paper, we explore the ability to use cloud browsers as *open computation centers.* To do this, we propose the Browser MapReduce (BMR—pronounced *beemer*) architecture, which is motivated by MapReduce [14], but contains distinct architectural differences. In our architecture, a master script running on a PC parameterizes and invokes *map-*

[1]Since JavaScript is Turing-complete, any computation is theoretically feasible.

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

per jobs in separate cloud browser rendering tasks. When complete, these workers save their state in free cloud storage facilities (e.g., provided by URL shortening services), and return a link to the storage location. The master script then spawns *reducer* jobs that retrieve the intermediate state and aggregate the mapper results.

To demonstrate the functionality of our cloud browser-based computational infrastructure, we implement three canonical MapReduce programs: *a*) word count, *b*) distributed grep, and *c*) distributed sorting. Compared to Amazon's Elastic MapReduce (EMR), BMR was faster for distributed grep, but several times slower for word count and distributed sort due to high communications costs. However, it does so at no monetary cost. Note that due to ethical considerations, we executed relatively small-scale computations in order to not overly tax the cloud browsers or the URL shortening services. As such, our experiments show a savings of only a few cents. However, larger jobs over longer periods of time can lead to substantial savings. Additionally, we explore the potential of BMR for performing parallelizable tasks that benefit from anonymity, e.g., cracking passwords. Attackers have already paid to use Amazon EC2 [2], and moving such activities to cloud browsers is likely.

This paper makes the following contributions:

- *We identify a source of free computation and characterize the limitations of four existing cloud browsers.* To our knowledge, we are the first to consider cloud-based Web browsers as a means of performing arbitrary computation.
- *We design and implement BMR, a MapReduce motivated architecture for performing large jobs within cloud browsers.* Cloud browser providers artificially limit computation to mitigate buggy Web pages. Using a MapReduce motivated architecture, we show how to coordinate resources in multiple cloud browser rendering tasks through the use of free storage made available by URL shortening services.
- *We port three existing sample MapReduce example applications to BMR and characterize their performance and monetary savings.* BMR has different limitations than traditional MapReduce (e.g., storage), and therefore must be optimized accordingly. We report on our experiences working within these limitations.

The remainder of this paper proceeds as follows. Section 2 overviews our architecture and lays out our design challenges. Section 3 characterizes the computation and memory limitations of several popular cloud browsers. Section 4 describes the BMR map and reduce primitives, scheduling, and the example applications. Section 5 discusses our implementation of BMR. Section 6 evaluates the performance of those examples in the Puffin browser. Section 7 discusses mitigations and optimizations. Section 8 overviews related work. Section 9 concludes.

2. APPROACH OVERVIEW

The goal of this paper is to explore methods of performing large computations within cloud-based Web browsers, ideally anonymously and at no monetary cost. While cloud browsers execute JavaScript code, which is Turing-complete, we expect cloud browser providers to implement resource limits on JavaScript execution. Therefore, we must divide

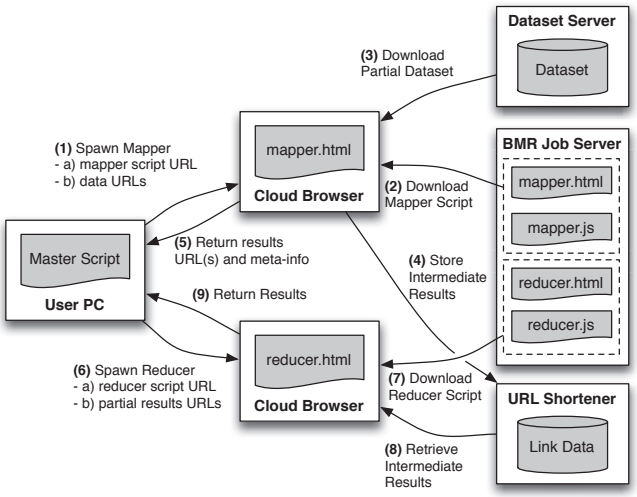


Figure 1: Browser MapReduce (BMR) Architecture

our large job into smaller parts. MapReduce [14] has become a popular abstraction for executing large distributed computation. Therefore, we propose a MapReduce-motivated execution framework called *Browser MapReduce* (BMR).

To better understand BMR, we first describe MapReduce. A MapReduce job is implemented as two procedures: *Map* and *Reduce*. Execution always begins with the mapping phase. A mapper extracts a set of key-value pairs of interest from each input record. For example, for a MapReduce job to count the number of words in a set of documents, the mapper determines the number of instances of each word in a small subset of the documents. Here, the word is the key and the number of instances is the value. The results of multiple mappers are then combined in the reducer phase. For word count, the reducer aggregates the word counts to produce an overall count for each word in the entire dataset.

In MapReduce, computational resources are abstracted as nodes within a cluster. Job coordination is performed by a *master* node. The master is responsible for handling communication synchronization, fault tolerance, and parallelization. Since a failure of the master node leads to a failed computation, the master node is often replicated. The remaining nodes in the cluster are worker nodes. A worker node can be a *mapper*, a *reducer*, or both. Note that a single MapReduce job consists of many mappers and reducers. To minimize communication overhead, the intermediate results generated by the mappers are stored locally and the locations are communicated to the master. By strategically partitioning the dataset, scheduling mapper and reducer jobs, and tracking these instances accordingly, the MapReduce framework can scale for both computationally intensive and large data processing applications.

Executing MapReduce operations within a cloud browser environment introduces several challenges:

- *Cloud browsers have artificial limitations.* Each cloud browser instance has artificial limitations placed on the amount of processing it can devote to a script, the size of memory allocated to that instance, and the time for which a script can execute on the browser safely without crashing the browser.
- *Job scheduling must account for the artificial limitations of the target cloud browser node* and *the expected*

compute time for the specific task. Each MapReduce application requires different complexity for the mapper and reducer. The scheduler must partition the job based on this complexity and the limitations of the target cloud browser.

- *Mappers cannot use local storage to communicate intermediate results.* There is no guarantee that a reducer can be spawned in the same cloud browser instance as the mapper. Furthermore, cloud browser instances cannot communicate with one another. Therefore, we must identify an alternative (ideally free) storage location for intermediate results.

The BMR architecture, shown in Figure 1, is motivated by the MapReduce framework, but differs in certain aspects. First, the master node runs on the user's PC. Since the master node is running on the user's PC, we can assume it is reliable. However, we must also assume it has limited bandwidth. Both the Map and Reduce functions are written in JavaScript and retrieved from the BMR job server hosting the application. To reduce the bandwidth requirements on the user PC, we assume that the dataset is statically served from a publicly accessible web server.[2] Due to the same-origin policy (SOP), if the data server is different than the script server, a CORS policy [29] must be set appropriately. Again, to reduce user PC bandwidth requirements, intermediate results are not returned directly to the master script. BMR uses a URL shortening service (e.g., `bit.ly`) for free cloud storage (see below). These URLs are then returned to the master script along with meta-information to aid reducer scheduling (discussed further in Section 4). Once the mappers complete, reducer scripts are similarly spawned and the final results returned. Note that the final results can be another set of URLs, or the data itself, depending on storage capacity limitations in the cloud browser instance.

As noted above, BMR requires cloud storage for intermediate results. We considered many alternatives for storing this data, and decided on a URL shortening service, as it is both free and semi-anonymous (an account is required, but doing so only requires a valid email address). Fundamentally, URL shortening services provide key-value storage, where a shortened URL returns a long data string, which in our experiments can be up to 2022 characters. Other options we considered included the simple strategy of returning the intermediate result data directly to the mapper script; however this requires user PC bandwidth. Another option is to store the intermediate results back to the dataset server. However, we wanted to decouple dataset storage from intermediate results storage for several reasons, e.g., read-only content is easier/cheaper to host. A third option was to pay for cloud storage (e.g., Amazon S3 [6]); however, computation would no longer be free.

3. BROWSER RESOURCE LIMITATIONS

BMR executes jobs (i.e., mappers and reducers) as rendering tasks for a cloud browser. In practice, cloud browser providers limit the resources provided to each rendering task to limit the consumption of buggy JavaScript. In order to optimally partition the input data and schedule workers, BMR must take into account the limitations of the target cloud browser. For each of the studied cloud browsers, we characterized the JavaScript capabilities; Flash and Java applets were not supported and therefore not characterized.

[2]If authentication is desired, authentication tokens can be passed to the mapper and reducer scripts.

```
function cpu_benchmark() {
  for(i=0; i<n; i++) {
    if(i%m == 0) {
      document.getElementById
          ("var").innerHTML = "Reached "+(i);
    }
  }
}
```

Figure 2: Computation benchmark

In the following discussion, we consider CPU cycles, elapsed execution time, and memory consumption. We also considered persistent storage provided by HTML5; however, we found it to be substantially lower than the RAM available to a cloud browser instance.

3.1 Benchmarks

We use simple JavaScript functions to benchmark the cloud browser capabilities. Each benchmark is designed to isolate a specific characteristic. Our benchmarking procedure has two stages. First, we use a small reporting interval to incrementally discover the limit. Then, we confirm that the interval reporting does not affect the results by specifying a report interval just below the determined limit. For example, the cloud browser might terminate a process if it is unresponsive, and updating the display indicates activity.

3.1.1 Computation

The CPU resource limitations configured by the cloud browser provider impacts how much data should be allocated to each worker. To measure CPU capacity, we perform a tight loop and report the maximum number of iterations reached. While this is not a direct measurement of CPU cycles, it provides an approximation for comparison and scheduling parameterization.

Figure 2 shows our CPU cycle benchmark JavaScript function. The function assumes a global variables `n` and `m`. `n` is a positive integer (e.g., 1 billion) specifying the number of iterations to perform. `m` specifies the "printing" interval. As mentioned above, the interval reporting helps to identify the progress before the browser crashes. After a limit is determined, `m` is increased to the maximum reached value to ensure that the act of printing does not extend the computation limit. Finally, we note that printing a value takes CPU cycles itself. While changing `m` can affect the maximum number of iterations performed, our results are conservative.

3.1.2 Elapsed Time

While cloud browser providers likely limit CPU cycles, they also might limit the wall-clock time. Since BMR requires the worker to download and upload data to network servers, the scheduling algorithm must account for this. To characterize execution time limits, we perform the JavaScript equivalent of `sleep()`.

Figure 3 shows our elapsed time benchmark JavaScript function. Here, we use `time_benchmark()` as a callback function passed to JavaScript's built-in `setTimeout()` timer. Note that `time_benchmark()` is not recursive. Rather, it returns and is called again by the JavaScript runtime. Finally, the benchmark is parameterized by a global variable `n` in-

```
var time=0;
function time_benchmark() {
  document.getElementById
      ("var").innerHTML = "Time: "+time;
  time = time + n;
  var x = setTimeout(time_benchmark,n*1000);
}
```

Figure 3: Elapsed time benchmark

```
function memory_benchmark() {
  var arr=new Array();
  for(i=0; i<n; i++) {
    arr.push(i);
    if (i%m == 0) {
      document.getElementById
          ("var").innerHTML="Reached " + i;
    }
  }
}
```

Figure 4: Memory consumption benchmark

dicating the number of seconds to sleep on each iteration. The benchmark keeps track of the total time and reports it on each interval. For testing, we began with small `n` (e.g., 1 second) and repeated times with a much larger `n` (e.g., 1 hour). Note that since some cloud browsers continued for hours without termination (see Section 3.3), we did not confirm a final maximum execution time.

3.1.3 Memory

The last resource limitation we characterize is working memory (i.e., RAM). Since BMR jobs operate on partial datasets, the scheduling algorithm needs to account for resident memory capacity when partitioning the dataset. Note that we also considered persistent storage (e.g., HTML5's LocalStorage); however, the W3C specification[3] indicates an arbitrary limit of 5MB per origin, which is significantly less then the memory capacities we report in Section 3.3. We verified the small persistent storage capacity on each of our target cloud browsers using a publicly available benchmark.[4]

Figure 4 shows our memory benchmark JavaScript function. This benchmark simply continues to append values to a dynamically allocated array. Similar to the CPU benchmark, the memory benchmark is parameterized by global variables `n` and `m` that determine the maximum number of iterations, and the reporting interval, respectively. Note that `n` also defines the upper bound on memory allocation. Since `arr` is an integer array, the memory capacity in bytes is $\texttt{i} \times 4$.

3.2 Studied Cloud Browsers

We analyzed the resource limits of the following commercially available cloud browsers. When selecting cloud browsers for BMR, one must ensure that the framework is capable of executing JavaScript on the server. For example, even though Opera Mobile uses Opera Turbo, we found that Web pages were compressed and rendered locally. This was also the case for the Android version of UC Browser [28].

Amazon Silk: The Amazon Silk browser [5] is exclusively available for Amazon's Kindle Fire tablet. Every time the user loads a web page, Silk dynamically decides whether rendering should occur on the device or on Amazon Web Services. Due to the dynamic nature of Amazon Silk's rendering decision, we were unable to confirm without doubt that the JavaScript was executed on the server for our experiments. However, we did observe the device communicating with Amazon continually through each experiment.

Cloud Browse: The Cloud Browse browser [3] is developed by AlwaysOn Technologies. It hosts a Firefox browser session on cloud servers and relays the rendered page to the mobile device. Cloud Browse currently only exists for iOS, but the Android version is expected. Since Firefox runs on the server, and based on observing continuous communication with Amazon EC2 servers throughout each experiment, we conclude that JavaScript executes on the server.

Opera Mini: The Opera Mini browser [23] is designed specifically for mobile environments with limited memory and processing power. It uses the Opera Turbo technology for faster rendering and compression of web pages. Note that the Opera Mobile browser also allows the user to enable Opera Turbo; however, in our experiments, enabling Opera Turbo only added compression and did not appear to render the content on the server. In contrast, the Opera Mini experiments where highly indicative of JavaScript execution on the server. The browser communicated to the server throughout the experiment. Furthermore, when the computation limit was exceeded, Opera Mini navigated to a Web page with the URL "`b:D8EAD704Processing`." This page had the title "Internal server error" and indicated "Failed to transcode URL" within the main window.

Puffin: The Puffin browser [12], developed by CloudMosa Inc., is designed for mobile devices and is advertised as rendering web pages in the cloud. It is available for both Android and iOS. The developer indicates that the browser does not store users' private data (e.g., cookies and history) on the cloud servers. All communication between the client and cloud browser is encrypted via SSL. During our experiments, we observed continuous communication between the client and cloud browser servers. Server side JavaScript execution is further confirmed by our implementation of BMR using Puffin in the latter sections of this paper.

3.3 Benchmark Results

We experimentally determined a conservative lower bound on the computation, time, and memory limits for each of the four cloud browsers. Amazon Silk was tested using version 1.0.22.7_10013310 on a Kindle Fire tablet. Cloud Browse was tested using version 4.2.1 (33) on an iPhone 3G running iOS v. 5.1 (9B176). Opera Mini was tested using v. 7.0.3 on a Samsung Galaxy Nexus running Android v. 4.0.2. Finally, we tested Puffin v. 2.2.5065 on a Samsung Galaxy Nexus running Android v. 4.0.2. The experiments for the reported results were performed in late May 2012.

Each browser has different failure modes, which lead to different strategies for determining the reported limit. Recall that the computation and memory benchmarks are parameterized by global variables `n` and `m`. Ideally, when the browser reaches its limit, it crashes in such a way that the greatest multiple of `m` reached is displayed. For Amazon Silk, a dialog box is shown, but the values on the Web page are still viewable. For Cloud Browse and Puffin, the computation simply stalls when the limit is reached, and no error message is reported. Finally, as described above, Opera Mini redirect to a server error page. In this case, we resorted to

[3] `http://dev.w3.org/html5/webstorage/#disk-space`
[4] `http://arty.name/localstorage.html`

Table 1: Cloud browser benchmark results

Browser	Computation Iterations	Computation ≈ Time	Elapsed Time	Memory Array Size	Memory Data Size
Amazon Silk	140,000,000	30 secs	24 hrs*	16,000,000	61 MB
Opera Mini	50,000,000	7 secs	6 secs	33,525,000	128 MB
Cloud Browse	40,000,000,000	1 hr	24 hrs*	121,000,000	462 MB
Puffin	200,000,000,000	2 hrs	24 hrs*	58,850,000	224 MB

* The benchmark was terminated after 24 hours.

using a binary search strategy to find the limit by adjusting `n` and keeping `m = n - 1`.

Computation: Table 1 shows both the number of iterations of the `for` loop that can be computed before the cloud browser fails. Our tests increase loops in increments of 1,000,000. We confirm the calculated value by setting `n` to that value and re-running the experiment 20 times. Table 1 also reports an approximate time for each experiment. The time varied per execution, but not significantly with respect to the listed duration.

Elapsed Time: The next column in Table 1 shows elapsed execution time when performing negligible computation (i.e., using `setTimeout()`). We let Amazon Silk, Cloud Browse, and Puffin run for 24 hours before terminating the experiment. Clearly, these browsers do not have a limit purely based on wall-clock time. However, Opera Mini consistently terminated after exactly six seconds. This is notable since computation benchmark experiments consistently exceed six seconds, indicating that Opera Mini terminates JavaScript execution if it is not performing computation.

Memory: The final two columns of Table 1 show the results of the memory benchmark. Using the appropriate search strategy, we discovered the reported value in increments of 500,000. Similar to the computation benchmark, we validated the value by re-executing the experiment a total of 20 times. For Opera Mini, we initially determined 34,000,000 array elements; however, after a few re-executions, this value failed. Reducing the limit to 33,500,000 for the subsequent runs succeeded. The listed value is the average of 20 trials. The memory for Puffin varied more greatly. When the memory limit of Puffin is exceeded, the display shows the last `m` value reached. Since updating the page does not affect Puffin runtime, we executed the experiments with `m` set to 500,000. The array size ranged from 50,000,000 to 66,500,000. We report the average in Table 1. This ranges indicates a dependence on either load on the server, or individual servers configured with different memory resource limits. Finally, the last column in the table simply converts the array size to bytes, assuming each array element occupies four bytes, and rounds to the nearest MB.

Summary: Based on our experiments, Cloud Browse and Puffin provide significantly more computational ability than Amazon Silk and Opera Mini; each provide at least an hour of computation time. While Cloud Browse provides more RAM, the difference turns out to be inconsequential. As discussed in Section 2, BMR uses a URL shortening service for communication between workers. For this paper, we use the popular `bit.ly` service. We experimentally determined that `bit.ly` can encode long URLs up to 2022 characters in length but rate-limits requests to 99 per IP address per minute. These limitations on communication size limitation makes BMR best suited for CPU-intensive tasks, as opposed to storage heavy tasks. As Puffin provides a higher computation limit than Cloud Browse, we chose it for our proof-of-concept BMR implementation.

4. DESIGNING AND SCHEDULING JOBS

In the previous section, we characterized the resource limitations of four cloud browsers and found that two of the cloud browsers provide a significant amount of processing capability. In this section, we show how one can break up larger jobs to run within executions of JavaScript (i.e., requests for Web pages) on cloud browsers. Our experiments led to two guiding principles: 1) optimize jobs for higher worker computation loads; and 2) limit worker communication data size where possible. This leads to different optimizations than one might find in MapReduce. For example, a BMR mapper can be more complex if it reduces the data communication size.

To understand BMR's ability to provide MapReduce functionality, we explore three canonical applications:

- *Word count:* Counts the number of instances of each word in a set of input files. The output has an entry for each word and the number of times it appears.
- *Distributed grep:* Finds instances of a regular expression pattern in a set of input files. The output is the matching lines. In the case of BMR, we output the file and line number of each match.
- *Distributed sort:* TeraSort [22] is a popular benchmark for MapReduce frameworks that implements a simplified bucket sort in a distributed fashion. We use input from the `teragen` application, which provides 98 character records containing 10 characters for the key and 88 characters for the value. Our BMR bucket sort application outputs the sorted keys and the file number in which the records originated.

By first looking at how map and reduce abstractions are designed and jobs scheduled, we will better demonstrate how these applications are implemented within BMR.

4.1 Map and Reduce Abstraction

Both the mapper and reducer jobs are implemented as Web pages that include JavaScript to perform the desired functionality. Each BMR application consists of `mapper.html` and `reducer.html` Web pages that include `mapper.js` and `reducer.js`, respectively.

Figure 5 depicts the BMR mapper abstraction. The mapper execution begins with the master script using the cloud browser to visit the URL of the mapper Web page (`http://bmr_server/mapper.html`). BMR assumes that the input for the overall MapReduce application is separated into many small text files. When each mapper job is started, the mapper URL includes HTTP GET parameters specifying private data `pdat` (e.g., range parameters for dividing

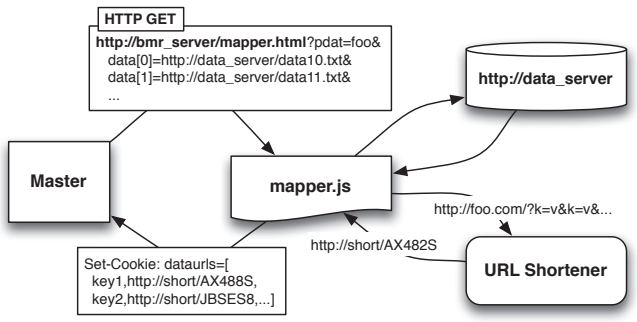


Figure 5: BMR mapper abstraction

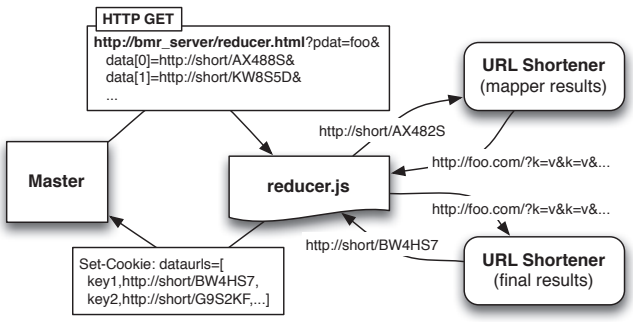


Figure 6: BMR reducer abstraction

data into URLs) and an array of the URLs of the input data for that worker. The mapper logic parses these HTTP GET parameters, downloads the corresponding files using `XMLHttpRequest`, and performs the desired map operation.

The mapper job communicates the intermediate map results to the master script using a URL shortening service (e.g., `bit.ly`). The map results are a set of key-value pairs. The BMR mapper encodes the key-value pairs as a "long URL", e.g., `http://foo.com/?k1=v1&k2=v2`. The long URL is sent to the URL shortening service in exchange for a short URL, e.g., `http://short/AX482S`. Because shorteners limit the number of characters in a long URL, the BMR mapper stores results across multiple URLs. Furthermore, the mapper might designate URLs as belonging to different partitions or buckets to help the master schedule reducers. Finally, the set of short URLs is returned to the master script along with a key value for each URL. The means of communicating the short URLs back to the master script varies per cloud browser platform. For the Puffin platform, we use the browser cookie field. More details of this design choice are discussed in Section 5.

Figure 6 depicts the BMR reducer abstraction. The BMR reducer is very similar to the BMR mapper. However, instead of retrieving its input from a data server, the inputs to the reducer are shortened URLs that are expanded to obtain the input data. In Figure 6, the final results are encoded as another set of long URLs. If the final results are small enough, they could easily be returned as cookie data.

Cross-Origin Requests: Both the mapper and reducer scripts perform `XMLHttpRequest` operations to retrieve and store data. JavaScript executing in a Web browser is subject to the *same-origin policy* (SOP), which prevents the script from making network requests to any origin (i.e., domain) except for the origin from which the JavaScript was downloaded. For example, if `mapper.js` is downloaded from `foo.com`, it cannot retrieve data from `bar.com`. This restriction can be overcome using cross-origin resource sharing (CORS) [29]. To allow `mapper.js` to request data from another origin, the Web server hosting the data must modify its HTTP headers to either explicitly allow scripts from the domain hosting `mapper.js`, or allow any domain (i.e., `*`). Note that `bit.ly` uses CORS to allow JavaScript running on any domain to perform network requests to its API.

4.2 Scheduling Jobs

Just as in MapReduce, BMR executes a mapper phase followed by a reducer phase. To effectively use cloud browser and URL shortening service resources, the master script must carefully partition the job. There are an arbitrary number of complex heuristics that one can use to tweak and optimize the master script scheduling. However, the goal of this paper is to explore how to perform computations within cloud browsers. Therefore, we assume: a) the input is divided into a large number of equally sized files that are accessible to the mapper and reducer JavaScript; and b) the following constants are specified by the BMR user:

f_s:	size of each input file (in bytes)
f_n:	number of input files
b_s:	maximum data size for cloud browser (in bytes)
u_s:	size of data in a shortened URL (in characters)
u_n:	number of shortened URLs a worker can create
α_m :	mapper compression factor for the BMR application ($\alpha_m > 0$)

Note that b_s must be empirically derived for the target cloud browser and the target BMR application; u_s is specific to both the BMR application and the URL shortening service; and α_m defines the compression factor from input file size to the output of the mapper (discussed below).

Mapper Scheduling: In the mapping phase, the master determines 1) the number of mappers to spawn, M_n, and 2) the number of input files to pass to each mapper, M_f.

Cloud browsers are limited in the amount of memory allotted to the worker. Scheduling must account for both the memory required to load the input data and the internal data structures to perform the processing. Because the total amount of required memory required by the worker is dependent on the specific BMR application, the BMR user must empirically determine b_s. We assume the input files are several times smaller than b_s (e.g., f_s is 2-5 MB).

As previously discussed, the key limiting factor is the number of shortened URLs that must be created. If the number of shortened URLs did not matter, the mapper scheduling is straightforward:

$$M_f = \left\lfloor \frac{b_s}{f_s} \right\rfloor \quad (1) \qquad M_n = \left\lceil \frac{f_n}{M_f} \right\rceil \quad (2)$$

However, as described above, URL shortening services such as `bit.ly` use rate limiting. It is therefore to our advantage to minimize the number of shortened URLs, since they are long-lived and take up part of the URL shortening service namespace. Recall that `bit.ly` can store 2022 characters per URL ($u_s = 2022$) and is limited to 99 URLs per minute. To avoid unnecessary delay, the BMR user can set $u_n = 99$; however, if multiple minutes wait in the mapper can be tolerated, u_n can be set higher.

Note that when choosing u_n, the BMR user must ensure that the mapper can transmit all of the shortened URLs

back to the master. Our proof-of-concept implementation of BMR using Puffin (Section 5) returns the shortened URLs in a cookie value. Puffin allows a maximum of 4053 characters in the cookie, and the identifying portion of `bit.ly` links is 10 characters, therefore u_n should be less than 400. Depending on the size of the key, even less links should be used. In the event that more links per worker are required, a tree structure of shortened links can be created; however, this results in extra processing.

Each BMR application has a different mapper compression factor α_m specified by the user. Typically, $\alpha_m > 1$, indicating that the output data is smaller than the input data. For example, our mapper for word count (described in Section 4.3) consolidates all instances of a word in the input text. We use $\alpha_m = 4.26$ for word count in Section 6. Using α_m, we redefine M_f as follows:

$$M_f = min\left(\left\lfloor \frac{b_s}{f_s} \right\rfloor, \left\lfloor \frac{\alpha_m \cdot u_n \cdot u_s}{f_s} \right\rfloor\right) \quad (3)$$

This modified equation accounts for the URL shortening service storage limitation.

Reducer Scheduling: The scheduling for the reducer phase is application specific. As a generic abstraction, we assume that the mapper stores key-value pairs into shortened URLs based on some partitioning strategy. For example, in word count, a partition is a range of characters, and in distributed sort, it is a bucket used in the bucket sort algorithm. When the mapper returns the set of URLs, it specifies a key for each URL. The master script schedules a reducer for each key, passing it all URLs corresponding to that key. Note that the number of keys or partition definition is passed as the private data (`pdat`) to the mapper and also affects the number of URLs used by the mapper (i.e., using partitions can lead to internal fragmentation).

4.3 Example Applications

To demonstrate the functional capability of BMR, we implement three canonical MapReduce applications: word count, distributed grep, and distributed sort.

Word Count: Word count determines how many times each word appears is a large set of text. This task lends itself well to the map and reduce abstraction. Traditionally, the word count map function parses part of the dataset, and for each word in the file, it prints, "`word: 1`". The reduce function tallies a count for each word in multiple files. Our BMR mapper behavior differs slightly. Since the BMR mapper must maintain the words in memory before "writing" them to the URL shortening service, it maintains a count for each word in the input files. We also include this reducer functionality within the BMR mapper to reduce the storage overhead of the intermediate results. To encode these results the BMR mapper creates a long URL similar to the following: `http://foo.com/?word1=5&word2=7&...`. As discussed in Section 4.2, the mapper partitions results into URLs to aid reducer scheduling. For simplicity, we use ranges of letters. For example, if three partitions are used, words are starting with a-h are in partition 1, i-p in partition 2, and q-z in partition 3. Note that multiple URLs will correspond to each partition.

Distributed Grep: Distributed grep performs a pattern match across many input files. In MapReduce, all of the work occurs in the mapper, and the reducer is simply the identity function. As such, our BMR implementation only requires a mapper; executing the reducer in a cloud browser provides negligible advantage. For distributed grep, the BMR mapper performs a pattern match on the input file. When a line i matches the pattern in file f, the mapper adds "`&f=i`" to the long URL. For example, if the mapper works on `bar1.txt` and `bar2.txt`, the resulting long URL will be encoded similar to the following: `http://foo.com/?bar1.txt=45&bar1.txt=48&bar2.txt=34`.

Distributed Sort: The popular TeraSort framework implements a distributed bucket sort. The keyspace is divided into n buckets (where n is provided as the `pdat` private data passed to the BMR mapper). The mapper sorts the input into the n buckets, but does not care about the order within the bucket. In the reducer phase, each reducer is given a bucket to sort. Since the buckets are ordered, the total ordering is obtained. For our experiments, we use input data generated by the `teragen` program included in the Hadoop framework (version 0.19.0). `teragen` produces 98 character records. Each record consists of a 10 character key and an 88 character value. Our BMR mapper only encodes the keys of the records due to the limited storage in shortened URLs. We store both the key and the file number in which the key originated, e.g., `http://foo.com/?key1=file1&key2=file2key3=file3`. Including the file number allows the master script to easily recombine the key with the value in post processing. Note that we assume files are sequentially numbered, therefore we only need to store the file number and not the entire filename.

Of the three applications, distributed sort has the greatest storage requirements. For each record in the input data, we store a 10 character key and several digits for the file number. Furthermore, the keys created by `teragen` include several non-alphanumeric characters that must be URL encoded, thereby occupying additional space. As such, the BMR user must define the compression factor α_m and number of buckets n accordingly. In Section 6, we use $\alpha_m = 2.513$ and $n = \left\lceil \frac{\text{No. of records}}{5000} \right\rceil$. By ensuring that reducer only shortens 5000 keys, we prevent issues with the `bit.ly` service, which is rate-limited to generating 99 URLs per minute per IP address, corresponding to slightly over 5000 keys.

5. IMPLEMENTATION

To demonstrate our ability to leverage computational resources from cloud-based Web browsers, we needed to have the ability to send these browsers to the URLs we desired, such that our jobs would be properly executed. Based on its overall high performance as shown in Section 3, we focused our efforts on the Puffin browser. Below, we describe how we adapted Puffin to work within the BMR framework.

In order to use Puffin within BMR, we required an understanding of how Puffin sends and receives messages, necessitating knowledge of how messages are generated and their format. Puffin is an Android application, so our first goal was to examine its operation to attempt to determine its message format. We used the *ded* decompiler [15] to convert the `.dex` file within the Android package into readable code. This allowed us to reconstruct the program flow, which we followed into the `libpuffin.so` library. From there, we used IDA Pro to dissassembled the ARM binary.

Puffin transmits its messages using SSL, thus simply intercepting messages using tools such as *wireshark* would not

be successful in allowing us to understand their format. To intercept traffic in the clear, we modified `libpuffin` to disable SSL certificate checking by inverting the logic for error check at the time of certificate validation. This allowed us to man-in-the-middle the connection with ease.

We wrote a parser after decompressing the cleartext in order to reverse the framing protocols. Individual messages form channels, which add flow semantics that make differing use of packed serialized objects and data dictionaries depending on the type of channel created. As an example, browsing a website leads to creation of a channel with a packed name-value pair object with name `service_id` and value *view*. Accessing cookie data registers a channel with `service_id` storing a value of *cookie_store*. Other channels are used for activities such as video streaming. Data directories are used with view channels to store additional information such as URLs and binary objects.

Puffin used an unusual encoding scheme with internal functions called *Q_encode* and *Q_decode*, though they have no relation to the standard Q-encoding scheme. The encoding appears to be an obfuscation measure that creates data larger than its corresponding plaintext. Characters are rotated deterministically and a counter added to their value before converting them into their hex representation, which is stored in ASCII. A checksum is included in a footer, likely to detect request tampering. Data is returned in a prerendered format from the Puffin servers.

Using this information that we gleaned through our analysis of the Puffin client, we wrote our own client that implemented the functionality required for connecting to the service. Our *Lundi* client creates channels for devices to connect to, a cookie store, and views for any URLs present. Because data is returned rendered as an image, we cannot scrape the page, but we do set cookies for operations that we perform and use those received cookies as intermediate data stores which can be stored as `bit.ly` links. The *Lundi* client is compact, written in under 900 lines of Python.

6. EVALUATION

In this section, we empirically evaluate the performance of the BMR system presented in the previous sections. We begin by describing our experimental setup. We then present a profile of the BMR results and a comparison to Amazon's Elastic MapReduce (EMR) and Hadoop on Amazon EC2.

6.1 Experimental Setup

To evaluate BMR with the word count, distributed grep, and distributed sort applications described in Section 4.3, we implemented a `mapper.js` and `reducer.js` library for each applications. The libraries consist over 1,000 lines of JavaScript. Both the JavaScript libraries and input data were hosted on the same Apache server for simplicity.

For each application, we performed tests on input data of sizes 1 MB, 10 MB and 100 MB. The input data was partitioned in multiple files, which varied per application. For word count, we downloaded the top 100 most downloaded books from `www.gutenberg.com/ebooks/`. To obtain 100 MB input, we downloaded additional books from the Top 100 most downloaded authors. For the experiments, the book text was concatenated and split into files of 1 MB in size. For distributed grep, we downloaded 140 MB of IRC logs for the *#debian* channel on freenode.net, splitting the input into files of size 10 MB. We grepped for a specific user entering and exiting the channel. Finally, for distributed sort, we used the Hadoop `teragen` program to create random records. Due to BMR's limitation on `bit.ly` URL creation, we split the input data into 0.5 MB files.

Table 2: Profile of BMR on Example Applications

Experiment*	# M	U/M	# R	U/R	Time
W.C. 1 MB	1	64	8	9.625	164.633s
W.C. 10 MB	10	24.7	8	17	178.859s
W.C. 100 MB	100	17.66	8	39	899.003
D.G. 1 MB	1	1	-	-	17.701s
D.G. 10 MB	1	1	-	-	18.680s
D.G. 100 MB	8	1	-	-	26.596s
D.S. 1 MB	2	56	2	43	61.040s
D.S. 10 MB	20	58.8	20	42.05	279.390s

* W.C. = Word Count; D.G. = Distributed Grep; D.S. = Distributed Sort; M = Mappers; R = Reducers; U = URLs; U/M and U/R are averages.

As discussed in Section 4.2, the BMR user must specify a cloud browser data size, b_s, and a compression factor α_m. For all applications, we found $b_s = 1MB$ to be sufficient. For the word count α_m, we selected 10 books at random and performed word count to determine the reduction in text size. We used the average reduction: $\alpha_m = 4.2673$. For distributed grep, α_m is intuitively very large, as the user entering and exiting events are only a small portion of the IRC logs, and BMR only needs store the input file and line number. Therefore, since Equation 3 will not use α_m, we conservatively set $\alpha_m = 1000$. Finally, for distributed sort, we calculated a conservative upper bound on data compression. For each 98 character record in the input, the distributed sort mapper must store 39 characters: *a*) a 10 character key, which may expanded to 30 characters due to URI encoding; *b*) three characters to URI encode the "="; *c*) three characters to store the file number; and *d*) three characters to URI encode the "&" character that separates key-value pairs. Therefore, we used $\alpha_m = 2.513$.

We execute each mapper and reducer in a separate worker nodes (i.e., cloud browser server), up to 20 simultaneous nodes, after which mappers wait for an open worker node. To prevent rate limiting from the back-end Puffin render servers, we sleep for 5 seconds between launching map instances on new nodes. Finally, to address the rate limitations on `bit.ly` URLs during our experiments, mappers and reducers randomly choose from a pool of 72 `bit.ly` accounts.

6.2 Results

BMR Results: Table 2 enumerates the results of the BMR experiments for the three applications, listing the number of mappers and reducers needed, and the average number of URLs needed per mapper and reducer. Note that the experiments were only performed once to limit the total number of shortened URLs created; running them multiple times does not change the distribution of data into mappers and reducers. Due to the compression factor, distributed grep passed the smallest amount of data, and therefore required both the smallest number of mappers and reducers and completed the quickest. As one might expect, the URL based communication between workers is a significant bottleneck.

Finally, the distributed sort 100 MB experiment tested the limits of our data passing methods. To meet the `bit.ly` rate limitation for reducers, we configured 200 buckets ($n = 200$). Returning 200 `bit.ly` links from the mapper exceeded

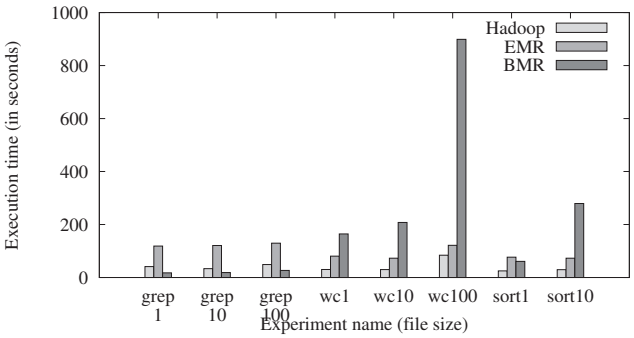


Figure 7: Comparison to Hadoop and EMR

the maximum cookie size. Therefore, alternative techniques such as a tree of `bit.ly` links are needed (see Section 4.2).

Comparison to MapReduce: To provide an intuition of the performance of BMR, we executed the same jobs on both Amazon's Elastic MapReduce (EMR) and Hadoop running on Amazon EC2. These experiments were performed on a cluster of 11 32-bit m1.small instances, allocating 1 instance for the master and using the other 10 as workers. Each m1.small instance was allocated 1.7 GB RAM, 1 EC2 Compute Unit, and 160GB storage. For EMR, data was stored in an Amazon S3 bucket, and for Hadoop, it was stored using Hadoop DFS. We used the Hadoop 0.19.0 Java example applications. Finally, for EC2, each m1.small instance cost US$0.08 per hour, and for EMR it cost US$0.095 per hour.

Figure 7 shows the comparison between BMR, Hadoop, and EMR for our test data. For distributed grep, BMR surprisingly performed better than both Hadoop and EMR. This is likely due to the relatively small amount of communication that was required; only one mapper was needed for 1 MB and 10 MB. However, in the word count and distributed sort experiments where substantially more data was communicated between the map and reduce phase, BMR's performance suffered. However, one should note that BMR was not designed to outperform existing MapReduce frameworks. Given BMR's limitations, it performed rather well.

Finally, given our small experiments, the cost savings were small. Hadoop experiments individually cost less than US$0.03, and the EMR experiments peaked at just under US$0.04. However, when performed at much larger scale and over a long period of time, BMR can amount to significant savings.

7. DISCUSSION

Recommendations for Cloud Browser Providers: By rendering Web pages in the cloud, the providers of cloud browsers can become *open computation centers*, much in the same way that poorly configured mail servers become open relays. The example applications shown in this paper were an academic exercise targeted at demonstrating the capabilities of cloud browsers. There is great potential to abuse these services for other purposes. We ran a series of hashing operations on the BMR infrastructure to determine how a password cracking implementation may be deployed and found with Puffin, 24,096 hashes could be generated per second, or 200 million per job. The infrastructure could be used for far more sinister purposes as well such as DoS and other amplification attacks, and pose ethical and possibly legal concerns. When deploying a cloud browser platform, providers should take care to place resource limitations on rendering tasks. As discussed in Section 3, two of the tested cloud browsers were capable of rendering for at least an hour. However, stricter resource limitations are not enough. A framework such as BMR can be used to link together rendering tasks into a larger computation. To mitigate such parallel jobs, providers should rate limit connections from mobile clients. The most primitive form of rate limiting is by IP address. However, NAT is used by some cellular providers, thereby making rate limitation by IP address impractical. As an alternative, users of cloud browsers should be required to create accounts, and rate limits should be placed on authenticated users. In our investigation of different cloud browsers, we observed that the Amazon Kindle Fire's Silk browser requires registration and sends a device-specific private key as part of its handshake protocol with the cloud-based renderers. Such a strategy is particularly helpful in mitigating the ability to clone instances. Additionally, existing techniques such as CAPTCHAs can limit the rate of creating new accounts.

Enhancing BMR: Our implementation of BMR was provided as a proof-of-concept. There are several aspects in which it could be improved. First, the use of `bit.ly` became an unintended bottleneck in the computation. We chose `bit.ly` due to its popularity as a URL shortening service as well as its easy to use APIs for creating short URLs. There are several alternative URL shortening services that could be substituted; however, these services likely have similar rate limits. Therefore, using a combination of URL shortening services may be more ideal. Furthermore, the use of shortened URLs may not be the best choice for applications that have a low compression factor (α_m). BMR simply needs some form of free key-value, cloud-based storage for communicating mapper results to the reducers. Services such as Pastebin (`pastebin.com`) can store significantly more than 2022 characters. However, account creation and rate limitation are still a concern. A second way BMR could be improved is scheduling. Our scheduling algorithms are relatively simple, and much more advanced scheduling strategies have been proposed for MapReduce [1]. Finally, BMR could be made to use multiple cloud browsers. Different cloud browsers have different benefits. For example, Puffin has more computational abilities than Cloud Browse, but Cloud Browse has more available memory. By using multiple cloud browsers, BMR could schedule mappers and reducers based on expected workloads.

8. RELATED WORK

Cloud computing creates a powerful new computing model but carries inherent threats and risks. Numerous studies [18, 11, 13, 25] have surveyed and examined security and privacy vulnerabilities in the cloud computing model from architectural, technical, and legal standpoints.

Ristenpart et al. [24] demonstrate that it is possible to map the internal cloud infrastructure and thus identify a particular VM of interest and spawn another VM as the co-resident of the target to mount cross-VM-side-channel attacks. Somorovsky et al. [27] perform security analysis pertaining to the control interfaces of both Amazon and Eucalyptus clouds, determining they can be compromised using

signature wrapping and XSS techniques. There have been multiple attempts to exploit these vulnerabilities. For example, Mulazzani et al. [21] successfully demonstrate an exploit of the popular Dropbox cloud service, analyzing the client and protocol to successfully test if a given file is present within Dropbox and consequently breaking confidentiality.

The large computation platform provided by cloud services such as Amazon's EC2 allows for large-scale password hashing and cracking. Moxie Marlinspike's CloudCracker service uses cloud services for cracking WPA passwords and other encryption types [20], while Amazon's GPU clusters have been used to crack 6-character passwords in under one hour [26]. Both of these services require payment to the cloud provider; BMR is the first service to show how free computation can be exploited through cloud browsing clients to perform arbitrary operations.

9. CONCLUSION

This paper investigated the ability to use cloud based Web browsers for computation. We designed and implemented a Browser MapReduce (BMR) architecture to tie together individual rendering tasks, then designed and executed three canonical MapReduce applications within our architecture. However, these example applications were simply an academic exercise to demonstrate the capabilities of cloud browsers, and form a preliminary investigation into a new way of performing parasitic computing. Based on our findings, we observe that the computational ability made freely available by cloud browsers allows for an *open compute center* that is valuable and warrants substantially more careful protection.

Acknowledgements

This work is supported in part by the National Science Foundation under awards CNS-1118046 and CNS-1222680, as well as in part by the U.S. Army Research Office (ARO) under grant W911NF-08-1-0105 managed by NCSU Secure Open Systems Initiative (SOSI).

10. REFERENCES

[1] A. Aboulnaga, Z. Wang, and Z. Y. Zhang. Packing the most onto your cloud. In *CloudDB*, Hong Kong, China, Nov. 2009.

[2] P. Alpeyev, J. Galante, and M. Yasu. Amazon.com Server Said to Have Been Used in Sony Attack. Bloomberg, May 2011.

[3] AlwaysOn Technoligies. Cloud Browse. `http://www.alwaysontechnologies.com/cloudbrowse/`.

[4] Amazon EC2 Pricing. `http://aws.amazon.com/ec2/pricing/`. Accessed April 2012.

[5] Amazon Inc. Amazon Silk FAQ's. `http://www.amazon.com/gp/help/customer/display.html/?nodeId=200775440`.

[6] Amazon Simple Storage Service. `http://aws.amazon.com/s3/`.

[7] A.-L. Barabasi, V. W. Freeh, H. Jeong, and J. B. Brockman. Parasitic computing. *Nature*, 412:894–897, 30 August 2001.

[8] S. Bjork, L. E. Holmquist, J. Redstrom, et al. WEST: A Web Browser for Small Terminals. In *ACM Symp. User Interface Software and Technology (UIST)*, 1999.

[9] O. Buyukkokten, H. Garcia-Molina, A. Paepcke, and T. Winograd. Power Browser: Efficient Web Browsing for PDAs. In *ACM SIGCHI*, pages 430–437, 2000.

[10] Y. Chen, X. Xie, W.-Y. Ma, and H.-J. Zhang. Adapting Web Pages for Small-Screen Devices. *IEEE Internet Computing*, 9(1):50–56, 2005.

[11] R. Chow, P. Golle, M. Jakobsson, E. Shi, J. Staddon, R. Masuoka, and J. Molina. Controlling data in the cloud: Outsourcing computation without outsourcing control. In *Proc. ACM CCSW'09*, Chicago, IL„ 2009.

[12] CloudMosa. Introducing Puffin. http://www.cloudmosa.com.

[13] K. Dahbur, B. Mohammad, and A. B. Tarakji. A survey of risks, threats and vulnerabilities in cloud computing. In *Intl. Conf. Intelligent Semantic Web-Services and Applications*, pages 12:1–12:6, 2011.

[14] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[15] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A Study of Android Application Security. In *USENIX Security Symp.*, San Francisco, CA, USA, Aug. 2011.

[16] R. Floyd, B. Housel, and C. Tait. Mobile Web Access Using eNetwork Web Express. *IEEE Personal Communications*, 5(6), 1998.

[17] A. Fox, I. Goldberg, S. D. Gribble, et al. Experience With Top Gun Wingman: A Proxy-Based Graphical Web Browser for the 3Com PalmPilot. In *IFIP Middleware Conference*, 1998.

[18] B. Grobauer, T. Walloschek, and E. Stocker. Understanding cloud computing vulnerabilities. *IEEE Security and Privacy*, 9(2):50–57, Mar. 2011.

[19] R. Han, P. Bhagwat, R. LaMaire, T. Mummert, V. Perret, and J. Rubas. Dynamic Adaptation in an Image Transcoding Proxy for Mobile Web Browsing. *IEEE Personal Communications*, 5(6), 1998.

[20] M. Marlinspike. Cloudcracker. `https://www.wpacracker.com`, 2012.

[21] M. Mulazzani, S. Schrittwieser, M. Leithner, M. Huber, and E. Weippl. Dark Clouds on the Horizon: Using Cloud Storage as Attack Vector and Online Slack Space. In *Proc. 20th USENIX Security Symposium*, San Francisco, CA, Aug. 2011.

[22] O. O'Malley. Terabyte sort on apache hadoop. `http://sortbenchmark.org/Yahoo-Hadoop.pdf`, May 2008.

[23] Opera mini & opera mobile browsers. `http://www.opera.com/mobile`.

[24] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *CCS*, 2009.

[25] J. C. Roberts II and W. Al-Hamdani. Who can you trust in the cloud?: A review of security issues within cloud computing. In *Info. Sec. Curriculum Development Conf.*, Kennesaw, GA, 2011.

[26] T. Roth. Cracking Passwords In The Cloud: Amazon's New EC2 GPU Instances. `http://stacksmashing.net/2010/11/15/cracking-in-the-cloud-amazons-new-ec2-gpu-instances/`, 2010.

[27] J. Somorovsky, M. Heiderich, M. Jensen, J. Schwenk, N. Gruschka, and L. L. Iacono. All your clouds are belong to us: security analysis of cloud management interfaces. In *ACM CCSW'11*, 2011.

[28] UCWeb. UC Browser. `http://www.ucweb.com/English/UCbrowser/patent.html`.

[29] W3C. Cross-Origin Resource Sharing. `http://www.w3.org/TR/cors/`, Apr. 2012. WD 3.

[30] X. Xiao, Q. Luo, D. Hong, H. Fu, X. Xie, and W.-Y. Ma. Browsing on Small Displays by Transforming Web Pages into Hierarchically Structured Sub-PAGES. *ACM Trans. Web*, 3(1):4:1–4:36, Jan. 2009.

Iris: A Scalable Cloud File System with Efficient Integrity Checks

Emil Stefanov*
UC Berkeley
emil@cs.berkeley.edu

Marten van Dijk
RSA Laboratories
mvandijk@rsa.com

Ari Juels
RSA Laboratories
ajuels@rsa.com

Alina Oprea
RSA Laboratories
aoprea@rsa.com

ABSTRACT

We present Iris, a practical, authenticated file system designed to support workloads from large enterprises storing data in the cloud and be resilient against potentially untrustworthy service providers. As a transparent layer enforcing strong integrity guarantees, Iris lets an enterprise tenant maintain a large file system in the cloud. In Iris, tenants obtain strong assurance not just on data integrity, but also on data freshness, as well as data retrievability in case of accidental or adversarial cloud failures.

Iris offers an architecture scalable to many clients (on the order of hundreds or even thousands) issuing operations on the file system in parallel. Iris includes new optimization and enterprise-side caching techniques specifically designed to overcome the high network latency typically experienced when accessing cloud storage. Iris also includes novel erasure coding techniques for the first efficient construction of a dynamic Proofs of Retrievability (PoR) protocol over the entire file system.

We describe our architecture and experimental results on a prototype version of Iris. Iris achieves end-to-end throughput of up to 260MB per second for 100 clients issuing simultaneous requests on the file system. (This limit is dictated by the available network bandwidth and maximum hard drive throughput.) We demonstrate that strong integrity protection in the cloud can be achieved with minimal performance degradation.

1. Introduction

Organizations that embrace cloud computing outsource massive amounts of data, as well as workloads to external cloud providers. Cost savings, lower management overhead, and rapid elasticity are just some of the attractions of the cloud.

But cloud computing entails a sacrifice of control. Tenants give up configuration and management oversight of the infrastructure that contains their data and computing resources. In cloud storage systems today, for example, tenants can only discover corruption or loss of their data (particularly infrequently accessed data) if their service providers faithfully report failures or security lapses—or when a system failure occurs. This integrity-measurement gap creates business risk and complicates compliance with regulatory requirements.

We propose a cloud-oriented *authenticated file system* called Iris that gives tenants efficient, comprehensive, and real-time data-integrity verification. The Iris system enables an enterprise tenant—or an auditor acting on the tenant's behalf—to verify the integrity and freshness of any data retrieved from the file system while performing typical file system operations. Data integrity ensures that data has not been accidentally modified or corrupted, while freshness ensures that the latest version of the data is always retrieved (and thus prevents rollback attacks reverting the file system state to a previous version). Moreover, tenants in Iris can efficiently audit the cloud provider on a regular basis and obtain continuous guarantees about the correctness and availability of the entire file system.

Motivating scenario We envision a scenario in which a large enterprise migrates its internal distributed file system to a cloud storage service. An important requirement for our system is that enterprise users (called herein *clients*) perform the same file system operations as they typically do (e.g., file read, write, update, and delete operations, creation and removal of directories) without modifying applications running on user machines. The slowdown in operation latency should be small enough to be unnoticed by users even when a large number of clients (on the order of hundreds and even thousands) issue operations on the file system in parallel.

Design goals in Iris Iris aims to support outsourcing of enterprise-class file systems to the cloud seamlessly and with minor performance degradation. Thus the design goals of Iris stem from the most common needs of enterprise-class tenants:

- *Efficiency:* Cloud file systems need to achieve throughputs close to those offered by local file systems under thousands of operations issued concurrently by many clients. Individual file system operation latency overhead should also be minimal.

- *Scalability:* A cloud file system should be scalable to large enterprise file systems under a variety of workloads with potentially very sensitive performance requirements. The system should also be scalable to multiple clients issuing operations on the file system in parallel.

- *Transparency:* Transparency and backwards compatibility with existing file system interfaces is important to facilitate migration to the cloud seamlessly.

*This research was mostly performed while visiting RSA Laboratories. Partially supported by an NSF Graduate Research Fellowship under Grant No. DGE-0946797 and a DoD National Defense Science and Engineering Graduate Fellowship.

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

- *Strong integrity protection:* Data and file system meta-data retrieved from the cloud need to be both *authentic* and *fresh*. Tenants' ability to verify continuously the integrity and availability of their data with minimal bandwidth and computation is a desirable feature, as well.

Contributions of Iris

In more detail, the key technical contributions and novel elements in Iris are:

- **Authenticated file system design:** The first contribution of Iris is to provide data integrity and freshness for an enterprise-class file system in an efficient way. To that end, we design a balanced Merkle-tree data structure that authenticates both file system data and meta-data blocks. The distinctive features of our data structure design compared to other authenticated file systems is that it efficiently supports updates from multiple clients *in parallel* (without blocking) and it handles *all existing* file system operations (including delete, move and truncate) with minimal overhead. Iris further implements many optimizations for typical file system workloads (e.g., those involving sequential file accesses).

In addition, Iris is designed to overcome the main economic barrier in migrating storage to the cloud: the impact of high network latency. Iris implements novel caching techniques locally, within the enterprise trust boundary. A lightweight (possibly distributed) trusted entity called *the portal* mediates file system operations passing between the enterprise clients and cloud and caches most recently accessed blocks. We develop techniques to cache the authentication information (nodes of the Merkle tree), handle dependencies among nodes, and preserve Merkle tree consistency when multiple clients simultaneously access nodes from the (partially cached) data structure.

- **Continuous auditing of file system correctness (PoR):** Iris provides the first construction for *dynamic* Proofs of Retreivability (PoR) [18]; it enables an enterprise tenant to continuously monitor the operation of the cloud storage service and obtain strong guarantees about the correctness and availability of the *entire file system*. With a PoR, a tenant can verify the correctness and availability of large data collection stored in the cloud with low computation and bandwidth cost. While previous PoR protocols are designed for static data (e.g., archival files), our protocol is the first to efficiently support *dynamic PoR* protocols over the entire file system. One of the key innovations in Iris is the design of a sparse randomized erasure code over the file system data and metadata. The new erasure code is specifically crafted to hide the code parity structure (typically revealed by other codes during file updates) and be resilient against a potentially adversarial cloud. It enables recovery when corruptions are detected through auditing.

- **End-to-end design and implementation:** One of our main contributions is the end-to-end design and full implementation of Iris consisting of 25,000 lines of code. We show through our performance evaluation that the caching mechanism in Iris is effective in achieving low latency for file system operations (similar to LAN latencies). Moreover, Iris achieves high throughput (up to 260MB for 100 clients issuing simultaneous requests on the file system in our local testbed), with the bottleneck given by the available network bandwidth and hard drive throughput. Finally, we demonstrate that the overall cost of adding strong integrity protection to Iris is minimal.

2. Related Work

File systems with integrity support: Early cryptographic file systems were designed to protect data confidentiality [6] and the integrity of data [29] in local storage. Later cryptographic networked file systems provided different integrity guarantees. TCFS [8] and SNAD [23] provide data integrity by storing a hash for each file data block. A number of systems construct a Merkle tree over files in order to authenticate file blocks more efficiently (e.g., [14, 13, 19, 4, 24, 25]).

Many cryptographic file systems to date provide data integrity, but do not authenticate the file system directory structure (or meta-data), e.g., [19, 24, 25]. Others, while authenticating both file system data and meta-data, do not provide strong freshness guarantees. SiRiUS [16] does not ensure data freshness, but only partial meta-data freshness by periodically requiring clients to sign meta-data entries. SUNDR [21] implements a property called "fork consistency" that detects freshness violations only when clients communicate out of band. More recently, SPORC [12] supports the building of collaborative cloud applications, enabling clients to recover from malicious forks performed by untrusted cloud servers. Depot [22] reconciles malicious forks even in the presence of faulty clients.

To the best of our knowledge, few cryptographic file systems provide freshness of both file system data and meta-data. SFSRO [14] and Cepheus [13] build a Merkle tree over the file system directory tree. While this approach efficiently supports file system operations like moving or deletion of entire directories, it results in an unbalanced authentication data structure and thus has a high authentication cost for directories with many entries. Athos [17] constructs a balanced data structure that maps the directory tree of the file system in a set of node relations represented as a skip list. Athos abstracts away the hierarchical structure of the directory tree, however, and doesn't provide efficient support for some existing file system operations, e.g., garbage collection. Moreover, its primary, prototyped design handles only a single client. FARSITE [4] is a peer-to-peer storage system that uses a distributed directory group to maintain meta-data information. Meta-data freshness is guaranteed when more than two thirds of the directory group members are correct. Data freshness is provided by storing hashes of file Merkle trees in the directory group.

Other systems provide data integrity guarantees for key-value stores. Venus [28] implements strong consistency semantics for a key-value store with malicious storage in the back-end. CloudProof [26] provides a mechanism for clients to verify the integrity and freshness, as well as other properties of cloud-stored data.

PoRs/PDPs: A *Proof of Retrievability* (PoR) [18] is a challenge-response protocol that enables a cloud provider to demonstrate to a client that a file is retrievable, i.e., recoverable without any loss or corruption. *Proofs of data possession* (PDP) [5] are related protocols that only detect a large amount of corruption in outsourced data. Most existing PDP [5] and PoR [18, 27, 7, 10] protocols are designed for static data, i.e., infrequently modified data.

Dynamic PDP protocols have been proposed by Erway et al. [11], but they were not designed to handle typical file system operations. For instance, Erway et al. [11] support operations like insertion in the middle of a file, but do not efficiently support moving and deleting entire files or directories. The CS2 system [20] designs and implements an efficient dynamic PDP protocol, as well as techniques for searching over encrypted data.

Several papers ([30] and [31]) claim to construct dynamic PoRs, but in fact only provide dynamic PDP schemes. To the best of our knowledge, designing efficient dynamic PoR protocols is extremely challenging and has stood as an open problem in the community.

3. System model and overview

Iris is designed as an enterprise file system using back-end cloud storage. Clients in Iris (enterprise users) issue file system operations intermediated by Iris and relayed to the public cloud. An important design consideration is that heavy caching on the enterprise side is strictly necessary. There are several reasons for this. First, if local caching is not performed, the cost of network transfer to and from the cloud will far outweigh any storage costs savings ([9] points to the extremely high cost of network transfer). Second, without local caching individual operation latency will be prohibitive for the system to be usable.

Existing network file systems are not designed with similar requirements in mind. For instance, NFS is not optimized for high network latency scenarios [15]. Moreover, most cloud storage systems available today (e.g., Amazon S3) export a key-value store interface and employ a flat namespace. Our system is unique in providing a file system interface to enterprise clients (for compatibility with existing applications), and at the same time ensuring low operation latency. In addition, our main goal is to support integrity protection of both file system data and meta-data and continuous verification of full file system correctness and availability with minimum overhead.

We describe here Iris's architecture, threat model, and give an overview of our solution and technical challenges.

3.1 System architecture

In our architecture (shown in Figure 1), a trusted portal residing within the enterprise trust boundary intermediates all communication between enterprise clients and the cloud. The portal caches data and meta-data blocks recently accessed by enterprise clients. Cached blocks are evicted once the cache is full and they are not utilized by a pending operation. The portal is also responsible for checking data integrity and freshness for all file system operations (with the *integrity layer* component). Data integrity ensures that data retrieved from the cloud has been written by authorized clients and has not been accidentally modified or corrupted at the cloud side. A stronger property, data freshness, ensures that data accessed by a client during a file system operation is always the latest version written to the cloud by any client.

The portal offers a *portal service* to clients issuing file system operations, and communicates to the cloud through the *storage interface* component. The auditing component issues challenges to the cloud periodically to verify the correctness and availability of the entire file system. The portal plays a central role in recovering from data corruptions: The portal caches error-correcting information (or more concisely, *parities*) for the full file system. When corruption is detected through the auditing protocol, these parities enable recovery of lost or corrupted data. Parities are backed up to the cloud on a regular basis (e.g., once a day or once a week).

To scale to large organizations with tens of thousands of clients, the portal needs to be distributed internally using a tool to ensure consistency of distributed caches (e.g., memcached [3]). For purposes of our prototype detailed in Section 6, we have instantiated the portal on a single server machine and show that it can scale up to 100 clients simultaneously executing sequential workloads in parallel on the file system.

The cloud maintains the distributed file system, consisting of all files and directories belonging to enterprise users. Iris is designed to use any existing cloud storage system transparently in the back end without modification. In addition, the cloud also stores the MACs and Merkle tree necessary for authenticating data, as well as the checkpointed parity information needed to recover from potential corruptions at the portal. As an additional resilience measure, the parity information could be stored on a different cloud or replicated internally within the enterprise.

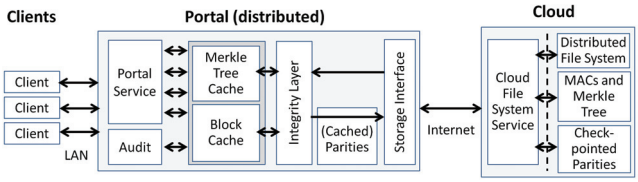


Figure 1: System architecture.

3.2 Threat model

Iris treats the portal, which is controlled by the enterprise, as a trusted component, in the sense that it executes client file system operations faithfully. No trust assumption is required on clients: They may act arbitrarily within the parameters of the file system. (The file system may enforce access-control policies on clients through the portal, but such issues lie outside the scope of Iris.)

The cloud, on the other hand, is presumed to be potentially untrustworthy. It may corrupt the file system in a fully Byzantine manner. The cloud may alter or drop file system operations transmitted by the portal; it may corrupt or erase files and/or metadata; it may also attempt to present the portal with stale, incorrect, and/or inconsistent views of file system data. The objective of the portal in Iris is to detect the presentation of *any invalid data by the cloud*, i.e., immediately identify any cloud output that reflects a file system state different from that produced by a correct execution of the operations emitted by the portal.

3.3 Solution overview and challenges

Iris consists of two major components:

Authenticated file system: As already described, the first challenge we address in building an authenticated enterprise-class file system is the high cost of network latency and bandwidth between the enterprise and cloud. Another challenge is efficient management and caching of the authenticating information. Integrity and freshness verification should be extremely efficient for existing file system operations and induce minimal latency.

Iris employs a two-layer authentication scheme. In its lower layer, it stores on every file block a message-authentication code (MAC)—generated by the portal when a client writes to the file system. These MACs ensure data integrity. To ensure freshness, it is necessary to authenticate not just data blocks, but also their *versions*. Each block has an associated version counter that is incremented every time the block is modified. This version number is bound to the file-block's MAC: To protect against cloud replay of stale file-blocks (rollback attacks), the counters themselves must be authenticated.

The upper layer of the authenticated data structure in Iris is a balanced Merkle-tree-based structure that protects the integrity of the file-block version counters. This data structure embeds the file system directory tree, and balances each directory for optimization. Attached to each node representing a file is a sub-tree containing file-block version counters. The root of the Merkle tree stored at the portal guarantees the integrity and freshness of both data and meta-data in the file system.

This Merkle-tree-based structure has two distinctive features compared to other authenticated file systems: (1) *Support for existing file system operations:* Iris maintains a balanced binary tree over the file system directory structure to efficiently support existing

file system calls; and (2) *Support for concurrent operations:* The Merkle tree supports efficient updates from multiple clients operating on the file system in parallel. Iris also optimizes for the common case of sequential file-block accesses: Sequences of identical version counters are compacted into a single leaf. We detail the data structure in Section 4, and the Merkle tree caching mechanism in Section 6.

Auditing protocol: Iris enables the enterprise tenant to continuously monitor and assess the correctness and availability of the entire file system through the auditing protocol. The auditing protocol in Iris is an instantiation of a PoR protocol and, in fact, the first dynamic PoR protocol supporting data updates. Previous PoR protocols have been designed for static data (files that do not undergo modifications). In any PoR, the tenant samples and checks the correctness of random data blocks retrieved from the cloud to detect any large-scale data corruption. To recover from small-scale damage, parity information computed with an erasure code needs to be maintained over the data.

The main challenge in designing a dynamic PoR protocol is that the erasure code structure, i.e., mapping of data blocks to parity blocks, must be randomized to prevent an adversarial server from introducing targeted, undetectable file corruptions. File updates are most problematic as they partially reveal the code structure (in particular the parity blocks corresponding to updated file blocks). At the same time, file updates should be efficient and involve only a small fraction of parity blocks.

We overcome this challenge with two techniques. First, we design Iris to cache parity information locally at the portal (and only checkpoint it to the cloud at fixed time intervals). As the cloud does not perceive individual file updates, but only parity modifications aggregated over a long time interval, the cloud cannot easily infer the mapping from file blocks to parity blocks. Second, we design a new sparse, binary code structure that combines randomly chosen blocks from the file system into a codeword. The code supports updates to the file system very efficiently through binary XOR operations. Its sparse structure supports very large file systems. This novel code construction is carefully parameterized to optimize local storage at the portal side, update cost, and bandwidth and computation in the auditing protocol. We describe the auditing protocol and the erasure code construction in Section 5.

4. Authentication in Iris

We describe in this section how Iris provides strong data protection, including integrity and freshness, for both file system data and meta-data. The authentication scheme in Iris is based on Merkle trees, and designed to support existing file system operations. In addition, random access to files for both read and write operations is a desirable feature (offered by existing file systems like NFS) that we also choose to implement. The tenant needs to maintain at all times the root of the Merkle trees for checking the integrity and freshness of data retrieved from the cloud. For reducing operation latency, recently accessed nodes in the tree are also cached at the portal (the caching mechanism is described in Section 6).
Figure 2 depicts the main components of our tree-based structure used for authentication:

Block-level MACs: To provide file-block integrity, we store a MAC for each file block, and combine block MACs from the same file in a *MAC file*. We choose to store MACs for each file block (instead of a single MAC for each file) to support random accesses to files. Block MACs are computed by the portal when a client writes to the file system. For providing freshness, we need to bind a unique version number to each file block every time it's updated and include the version number in the block MAC. To protect against rollback attacks (in which clients are presented with an old state of the file system), version numbers will have to be authenticated as well.

File version trees: We construct a *file version tree* per file that authenticates version numbers for all file blocks in a compressed form. Briefly, the file version tree compresses the versions of a consecutive range of blocks into a single node, storing the index range of the blocks and their common version number. File version trees are optimized for sequential access to files. For instance, if a file is always written sequentially then its file version tree consists of only one root node. The compacted version tree essentially behaves as a range tree data structure. An example of a compacted tree is shown in Figure 3.

Directory trees: To authenticate file system meta-data (or the directory structure of the file system), the file system directory tree is transformed into a Merkle tree in which every directory is mapped to a *directory subtree*. We have chosen to map our authenticated data structure onto the existing file system tree in order to efficiently support file system operations like delete or move of entire directories. To support directories with large number of files efficiently, we create a balanced binary tree for each directory that contains file and subdirectory nodes in the leaves, and includes intermediate, empty internal nodes for balancing. Nodes in a directory tree have unique identifers assigned to them, chosen as random strings of fixed length. A leaf for each file and subdirectory is inserted into the directory tree in a position given by a keyed hash applied to its name and its parent's identifier (to ensure tree balancing).

At the leaves of the directory tree, we insert the file version trees in compacted form, as described above. Internal nodes in the Merkle tree contain hash values computed over their children, as well as some additional information, e.g., node identifiers, their rank (defined as the size of the subtree rooted at the node), file and directory names.

Our Merkle tree supports the following operations. Clients can insert or delete file system object nodes (files or directories) at certain positions in the tree. Those operations trigger updates of the hashes stored on the path from the inserted/deleted nodes up to the root of the tree. Deleted subtrees are added to the free list, as explained below. Clients can verify a file block version number, by retrieving all siblings on the path from the leaf corresponding to that file block up to the root of the tree. Searches of files or directories in the tree can also be performed, given absolute path names.

We also implement an operation randompath-dir-tree for directory trees. This feature is needed to execute the challenge-response protocols of the auditing component in Iris. A (pseudo)-random path in the tree is returned by traversing the tree from the root, and selecting at each node a child at random, weighted by rank. In addition, the authentication information for the random path is returned, so the tenant can verify that the path has been chosen pseudo-randomly.

With this Merkle tree construction, we authenticate both file system meta-data, as well as file block version numbers. Together with the file block MACs, this mechanism ensures data integrity and freshness, assuming that the portal always stores the root of the Merkle tree.

Free list: As an optimization, we also maintain in the data structure a *free list* containing pointers of nodes deleted from the data structure, i.e., subtrees removed as part of delete or truncate operations. The aim of the free list is to defer garbage collection of deleted nodes and support remove and truncate file system operations efficiently. We omit further details due to space limitations.

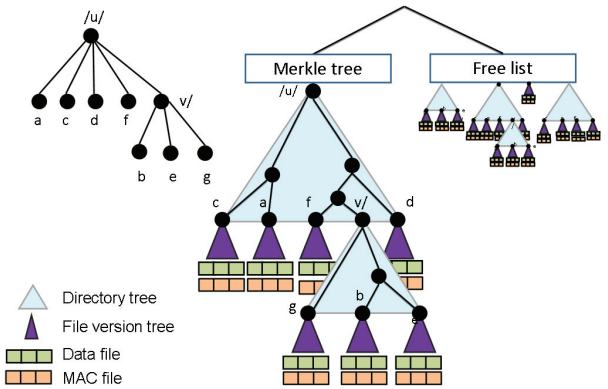


Figure 2: Authenticated tree. A file system directory on the left and its mapping to the Merkle tree on the right.

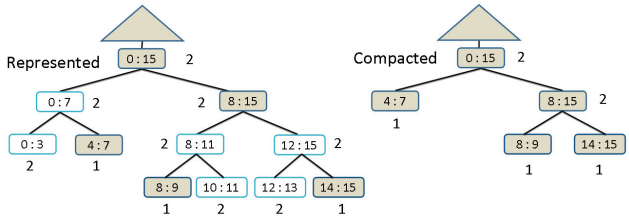


Figure 3: File version tree for a file with 16 blocks. Blocks 0-3 and 10-13 have been written twice, all other blocks have been written once. White nodes on the left are removed in the compacted version on the right. Version numbers are adjacent to nodes.

5. Auditing protocol

The authentication mechanism in Iris presented in the previous section can be used to verify the correctness of all blocks retrieved from the file system during the course of normal operations issued by clients. A challenging question that we address in this section is how can the enterprise verify infrequently accessed blocks and detect even small amounts of corruptions spread throughout the file system. We are particularly interested in offering strong assurances to the enterprise about the correctness and availability of *the entire file system*. An important requirement is that auditing of correctness should be performed with minimal bandwidth and computation. For instance, downloading a substantial fraction of the file system to verify its correctness would not be an acceptable solution. In addition, a recovery mechanism is needed to reconstruct the original data once corruptions are detected.

Several different protocols that address to some extent this question have been proposed in the literature. PoR protocols provide strong assurances about availability of data outsourced to the cloud, and a recovery mechanism, but they have only been designed for static data (files that do not undergo modifications). PDP protocols, while supporting updates to data, ensure only *detection* of a certain amount of data corruption, but do not implement a recovery mechanism. To the best of our knowledge, our solution here is the first *dynamic PoR protocol* over an entire file system, supporting updates and providing an efficient recovery mechanism in case data corruption is detected.

We start by presenting at a high level how existing PoR protocols work, and then describe the challenges of adapting these ideas to a dynamic setting. We then discuss our main insights and contributions in constructing a dynamic PoR protocol.

5.1 Static PoR protocols

In a PoR protocol, the tenant encodes a single file with an error-correcting code (ECC) and stores the encoded file in the cloud. The encoded file contains the original file and some *parity blocks*, redundant blocks computed with the ECC that are needed in recovering from corruption. To ensure correctness and availability of the data, the tenant periodically challenges the cloud for a few randomly selected file blocks, and verifies their correctness. Through this auditing protocol, the tenant can detect large-scale corruption to the file (exceeding a certain fixed threshold). Small corruptions, while not detectable through sampling, can be recovered from the redundancy embedded in the encoded file.

An important parameter in a PoR is the recovery-failure probability ρ. This is the probability, assuming that the cloud replies correctly to all challenges during an audit, that the tenant can't recover the file from the cloud's storage. The size and frequency of challenges in a PoR may be calibrated to achieve a target parameter ρ given the file size, and error-correcting code parameters.

5.2 Challenges for dynamic PoRs

The main challenge in adapting a static PoR protocol to a dynamic setting is the construction of an error-correcting code with several required properties. As a reminder, the error-correcting code is used to recover from corruptions once the auditing protocol detects missing or corrupted data at the cloud. An additional requirement our system has compared to previous PoR protocols is that it needs to recover from corruptions of both data and meta-data in the entire file system (while previous PoR protocols have been designed for single files).

Our first observation is that we can use in our system an erasure code instead of a more expensive error-correcting code. The reason is that Iris's main service is authentication of file system blocks, and, therefore, the portal can verify the correctness of file blocks and Merkle tree nodes during recovery and determine the positions of corrupted blocks. We present the remaining challenges in achieving an efficient dynamic PoR protocol:

Challenge 1: Update efficiency The erasure code has to support updates to the file system efficiently. In particular a modification to a file block or Merkle-tree node should require the update of only a small number of parity blocks. Additionally, it would be preferable to use cheap Galois field arithmetic in the parity computation, such as $GF(2)$ which essentially consists of XOR operations. Higher order Galois field arithmetic (as employed by Reed-Solomon codes, for instance) is too expensive to sustain our desired throughput of several hundred megabytes per second.

This requirement excludes upfront the use of maximum-distance separable (MDS) codes. While such codes are attractive for their correction capability, a parity block in an MDS codes depends on all message blocks, and therefore updates to the codeword are quite impractical.

Thus we must use a non-MDS code, with a lower error-correction capability. For instance, we might *stripe* the file system, that is, partition it into a number of smaller components, called stripes, and apply an erasure code individually to each stripe (striping is a common technique employed in most storage systems today). With this approach, updates would be more efficient as an update to a file block or Merkle tree node would involve updating only parity

blocks within a single stripe.

Challenge 2: Hiding code structure Nevertheless, striping introduces a problem. When a client updates a block of the file system along with the corresponding stripe parities, it *reveals code-structure information* to the cloud, namely the correspondence between the file blocks and the parity blocks. A malicious cloud can then create a targeted corruption against the file system, e.g., it can corrupt a single stripe and its corresponding parity blocks. Such corruption, being focused, will be hard to detect by sampling (since sampling detects only a large amount of corruption).

We overcome this challenge with two key techniques:

1. *Cache parities at the portal* We cache the parity information at the enterprise side and only transmit parities to the cloud at regular time intervals for back up (e.g., at the end of the week). With this approach, the cloud does not perceive individual updates to the file system, but only the aggregate parity structure over a large number of updates and can not infer the exact code structure. Moreover, updates are extremely efficient if parities are stored in main memory at the portal.
2. *Randomize code structure* Even when parities are stored at the portal, it is important that the stripe structure is not revealed to the cloud to avoid targeted corruptions. To enforce this, we randomize the assignment of file blocks to stripes.

If these two design principles are employed, it might seem that after caching the parities locally and randomizing the assignment of file blocks and tree nodes to stripes, any erasure code could be used for computing the parity blocks within a stripe. But our system has to overcome another subtle challenge:

Challenge 3: Variable-length encoding Typically, the code parameters for an erasure code, including the message size, and the size of parity information are fixed and known in advance (before encoding is performed). However in Iris we need to compute parity blocks over an entire file system data and meta-data blocks without knowing in advance the total size of the file system. At the same time, we have to enforce a randomization of the mapping of file system blocks to parity blocks at any given time. Therefore, approaches in which new parity blocks are created as more data is added to the file system in a streaming fashion (e.g., LDPC codes) would not be applicable here.

New sparse randomized erasure code construction. Our solution is to set an upper bound on the total size of the file system, and design a novel erasure code construction that is *sparse* in the sense that it supports incremental updates to the codeword very efficiently, even when only a fraction of the maximum size is used by the file system. The construction randomizes the mapping of file system blocks to parity blocks, and uses binary XOR operations. The size of the parity information is also constrained to fit into the main memory of typical servers today (an important consideration for efficient updates). We are able to prove for this construction an exponentially small bound for the recovery-failure probability.

If the file system needs to be expanded, the error correcting codes can be rebuilt, but a more bandwidth efficient solution would be to double the ECC data structure when the file system doubles in size.

5.3 Our erasure code

Parameter overview: We first set an upper bound for the entire file system size, denoted n. In our example parameterization, n is the number of 4KB blocks needed for a file system of size 1PB. Our erasure code construction is scalable up to that size, but once the file system exceeds the upper bound, the code parameters need to be changed and the file system has to be re-encoded.

To correct a fraction α of erasures, the storage for parities must be at least $s \geq \alpha n$ blocks—a coding-theoretic lower bound. Here s is limited by the sizes of current memories to about $s = O(\sqrt{n})$ for practical file system sizes and thus $\alpha = \Omega(1/\sqrt{n})$. (To obtain a probabilistic guarantee that at most an α-fraction of all stored file blocks is missing or corrupted, the tenant must challenge $c = O(1/\alpha) = O(\sqrt{n})$ randomly selected file blocks.)

To support updates efficiently we split the huge codeword into $m \approx \alpha n$ stripes; each stripe being a codeword itself with p parities. With high probability, given an α-fraction of erasures, each stripe is affected by only $O(\log n)$ erasures. Thus to correct and recover stripes, we need $p = O(\log n)$ parity blocks per stripe, leading to $s = O(\alpha n \log n) = O(\sqrt{n} \log n)$ memory. Each write only involves updating $u = O(\log n)$ parities within the corresponding stripe. By using a sparse parity structure, though, we are able to reduce u to $O(\log \log n)$.

Details on our erasure-code construction: Our erasure code is a sparse one based on efficient XOR operations. Although the new construction is probabilistic in that successful erasure decoding is not guaranteed for any number of erasures, its main advantage is that it is a binary efficient code scalable to large codeword lengths.

The portal computes parities over both file blocks and Merkle tree nodes when block values are updated by a client operation. For the purpose of erasure coding, we view data blocks or tree nodes as identifier-value pairs $\delta = (\delta_{id}; \delta_{val})$, where δ_{id} is a unique identifier (a unique block ID in the file system) and $\delta_{val} = (\delta_1, \ldots, \delta_b)$ is a sequence of b bits denoting the change in block value. (We assume all blocks are initialized with 0.) To randomize the mapping from data blocks to parity blocks, we use a keyed hash function $H_k(.)$ that maps an identifier δ_{id} to a pair (θ_{ind}, θ), where θ_{ind} is a random stripe index and $\theta = (\theta_1, \ldots, \theta_p)$ is a binary vector of p bits. The randomization is graphically depicted in the full version of this paper [1].

The 1s in vector θ indicate the parity bits that need to be updated. Each update modifies at most u of the p parities of the stripe to which δ belongs. That is, $H_k(\delta_{id})$ is designed to produce a binary random vector θ of length p with at most u entries equal to 1. For $u = O(\log p) = O(\log \log n)$ this leads to a sparse erasure code that still permits decoding, but entails relatively few parity updates.

Encoding: We maintain a parity matrix $P[i]$ for each stripe i, $1 \leq i \leq m$. To change the value of block δ_{id} with δ_{val}, the portal computes $H_k(\delta_{id}) = (\theta_{ind}; \theta)$; constructs $A = \delta_{val} \otimes \theta = \{\delta_i \theta_j\}_{i \in [1,b], j \in [1,p]}$; and updates $P[\theta_{ind}] \leftarrow P[\theta_{ind}] \oplus A$. The change in parity structure is shown graphically in the full version of this paper [1].

Since vector θ has at most u non-zero positions, the number of XOR operations for updating a block is u. The total storage for all parities is $s = bpm$ bits.

Decoding: Erasure decoding of the multi-striped structure involves decoding each stripe separately. Gaussian elimination is performed m times, each time computing the right inverse of a $(\leq p) \times p$ matrix–at a cost of at most $p^2 = O((\log n)^2)$ XOR operations. As an additional benefit of our construction, decoding can be done in place, and thus within memory at the portal.

Analysis: The full version of this paper [1] provides a detailed analysis. E.g., with a block size of 4KB, 5KB communication per challenged block, 5.8GB total communication per challenge-response round, 16GB of main memory at the portal for parity storage, and 1PB file system size, we achieve recovery failure probability $\rho \leq 0.74\%$.

5.4 Erasure-coding for Dynamic PoR

We now explain how our erasure code functions in Iris.

PoR encoding and update: During encoding, the portal constructs two parity structures: the *data parity structure* constructed over the file system data blocks (including the data blocks in the free list) and the *meta-data parity structure* over the meta-data blocks (internal nodes in the data structure comprising the Merkle tree and free list).

The challenge-response protocol: The portal challenges the cloud to return a set of c (again $c = O(\sqrt{n})$ randomly selected file system data blocks, including data blocks from the free list. These blocks are all leaf nodes in the authenticated data structure containing the Merkle tree and free list. As an optimization, the portal sends a seed from which the challenge set is derived pseudo-randomly.

The c selected random blocks together with the authenticating paths from the authenticated data structure are transmitted back to the portal. The portal verifies the correctness of the responses by performing two checks. First, it verifies the integrity and freshness of the selected blocks, checking the block MACs and the path to the root in the authenticated data structure. Second, it verifies that the blocks have been correctly indexed by the challenges according to the node ranks/weights. (This proves that the file system data blocks are selected with uniform probability.) As a byproduct of these checks the challenge-response protocol also verifies the integrity and freshness of the meta-data blocks (internal nodes in the authenticated data structure). We can immediately infer that if a fraction α of file system data blocks don't verify correctly, then at most a fraction α of internal nodes in the Merkle tree and free list are either missing or corrupted.

Recovery: See the full version of this paper [1] for the recovery algorithm.

6. Implementation

Our implementation of Iris is a 25,000-line end-to-end system with all integrity checking in place. The system is fully asynchronous and never holds a lock while waiting for network or disk I/O operations. The code runs in user space as a transparent layer that can take advantage of any existing storage system at the cloud provider. Our implementation uses the open-source .NET framework Mono, which is advantageously platform-independent: Iris can run on Linux, Windows, and MAC OS.

Our implementation includes the Portal, a simple Cloud storage server, and clients that run traces and benchmarks, as depicted in the detailed system architecture in Figure 1.

6.1 Cloud

The cloud stores not only regular file system data, but also authenticating meta-data, including MAC files and our Merkle tree authenticated data structure, as well as checkpointed parities needed for recovery. The repositories for these data types are shown at the top of Figure 1.

The portal performs reads and writes to the various data repositories by invoking their respective cloud-side services. The *Cloud File System Service* handles requests for file blocks, MAC files, and the Merkle tree (stored in our implementation in an NTFS file system). Operations on file blocks are executed asynchronously at the portal. Sequential access operations to the same file can potentially arrive out of order at the cloud. (Re-ordering can occur in transit on the network, as our portal and cloud machines are each equipped with three network cards.) To reduce disk spinning, the Cloud File System Service orders requests to the same file in increasing order by block offset.

6.2 Portal

The portal interacts with multiple clients which issue file system calls to the *Portal Service*. The portal executes client operations in parallel: Each operation is executed in a thread pool as a user-scheduled task with asynchronous steps. When an operation is waiting for a long running step such as disk or network I/O, the task is paused and the current thread switches to another task. This allows thousands of simultaneously active operations to be handled by the thread pool with a small number of threads. In our setup, the thread pool had 16 threads—one for each virtual CPU core, for maximum parallelism.

Operations don't interact directly with the cloud, but instead with the Merkle Tree and Block Caches. All data and meta-data requested by the caches is downloaded from the cloud via the *Storage Interface* in the portal. While in use by an active operation, blocks and nodes are retained in the cache. Prior to being cached, however, blocks and nodes downloaded from the cloud are checked for integrity by the *Integrity Checker* components.

Our implementation benefits from multi-core functionality available in most modern computers. Operations performed on active blocks in the cache are split into atomic operations (e.g., hash update for a tree node, check MAC for a data block or compact nodes in file version trees). These are inserted into various priority queues maintained at the portal. Multiple threads seize blocks from these queues, lock them and execute the atomic operations. Operations are always started in order, but may complete out of order. However, our implementation ensures that the effect of the operations on the system is the same as if they were executed by a single thread in order. If multiple clients issue conflicting operations simultaneously, they are executed in the order in which they arrive to the Portal. It is the responsibility of the clients to perform locking out-of-band.

Distributing the Portal. For scalability, the portal can be distributed across multiple machines. Each portal machine would then be responsible for a subtree of the file system. When clients first mount the file system, they can contact any one of the portals to get the assignment of portal machine to subtrees. As the file system changes over time, a subtree may grow or shrink substantially, and then the subtree assignment will need to be rebalanced by splitting a subtree and copying one part of it onto another portal. Our current implementation does not support this, but we would like to point out that even with a single portal, Iris can achieve a throughput of up to 260 MB/s, which already exceeds the bandwidth to the cloud for many enterprises. Additional challenges (e.g., caching to reduce latency) arise when the portal is *geographically* distributed, but these are out of the scope of this paper.

6.2.1 Merkle tree cache

The Merkle Tree Cache in the portal is Iris's most complex component. Much of the design effort and complexity of Iris lies in the caching strategy for recently accessed portions of the tree. We designed a generic, efficient Merkle Tree Cache that ensures consistency across thousands of simultaneous asynchronous client operations.

When an operation accesses the cache, it first locks it using a mutex and unlocks it when it's done. All of the operations are designed such that they access the cache for a very short period of time for tasks such as changing the value of a few fields of a Merkle tree node. To ensure a high degree of parallelism, the Merkle tree mutex is never locked while an operation waits for a long running step such as network or disk I/O.

When executing operations in parallel, a real challenge is to handle dependencies among tree nodes and maintain data structure

consistency and integrity. We do this by imposing several orderings of operations. Nodes are brought into the cache in a top-down order and are evicted in a bottom-up order. The top-down ordering is necessary because when a node is read from the untrusted storage, it can only be verified once all of its ancestors have also been cached in and verified. Likewise, a node can only be written out to the untrusted storage after the hash of its subtree has been computed. If multiple nodes in a sub-tree are modified, the Merkle Tree Cache will only hash the shared path to the root once, thereby significantly reducing the number of hashes that need to be performed.

Phases. To enforce the ordering, each node is always in one of the following phases: Reading, Verifying, Neutral, Compacting, UpdatingHash, or Writing. A node always traverses these phases in order and only after its parent or children have reached a certain phase. For example, a node only enters the verifying phase after its parent has completed the verifying phase. The Reading and Verifying phases are applied top-down and the Compacting, UpdatingHash, and Writing phases are applied bottom-up. When a node is in the Neutral phase, it is in the cache and available to be used by operations.

Pinning. Operations oftentimes need to access multiple nodes. For example, a WriteFile operation needs to access the path in the version tree that descends all the way to the version node corresponding to a specific block. The operations first *pin* all of the nodes they need and then proceed to execute. If a node is needed by an operation and is not currently in the cache, the operation is paused and resumed when all of its pinned nodes have been loaded into the cache. Once a node is pinned, it is not cached out until it is unpinned (e.g., when the operation completes). A node may be pinned multiple times, in which case it must be unpinned the same number of times until it is considered in the unpinned state and may be cached out.

If a node is pinned, its ancestors, sibling, and siblings of the ancestors are automatically *indirectly* pinned. This is necessary because if the node is modified, the indirectly pinned nodes will be needed when updating the hashes of the path to that node.

Eviction. When the cache reaches its maximum allowed size, it repeatedly evicts least-recently-used (LRU) leaf nodes, causing a bottom-up wave of evictions. Evicting a node consists of transitioning its phase from the Neutral to Compacting. The node then goes through the UpdatingHash and Writing phases until it is finally removed from the cache. If a node and its subtree were not modified, then the UpdatingHash and Writing phases are skipped. If all of the nodes are pinned, then new operations block until the currently executing operations complete and unpin more nodes.

6.2.2 Other components

The Block Cache functions much like the Merkle Tree Cache except that blocks don't have parents/children so there are no dependencies between blocks.

The Merkle Tree and Block Caches keep track of two items per node/block: The old and new data. The old data is the value of the node/block when it was fetched from the cloud. The new data is its value after it was (possibly) modified by an operation. When a node/block is evicted, the portal computes the difference of the byte representations of the old and new data and updates the parities.

Another component of the portal is the auditing module. This service, periodically invoked by the portal, transmits a PoR challenge to the cloud and receives and verifies the response, consisting simply of a set of randomly selected data blocks in the file system and their associated Merkle tree paths. The portal also maintains a repository of *Parities* to recover from file system corruptions detected in a PoR, seen in the portal cache module in Figure 1. Parities undergo frequent modification: Multiple parities are updated with every file-block write. Thus, the Parities repository sits in the main memory of the portal.

The portal can include a checkpointing service that backs up data stored in the main memory at the portal to local permanent storage. To enable recovery in the event of a portal crash, checkpointed data can be periodically transmitted to the cloud (with a MAC for integrity). While we have not implemented this component, it can rely on well-known checkpointing techniques.

7. Experimental evaluation

We ran several experiments to test different aspects of Iris. We first describe our setup and then present our results. Two machines ran the full end-to-end system implementation described in Section 6: The Portal and the Cloud.

Portal Computer. The Portal computer has an Intel Core i7 processor and 12 GB of RAM. The experiments were run on Windows 7 64-bit installed on a rotational disk, but no data was written to the Portal's hard drive for the purpose of our experiments.

Cloud Computer. The Cloud computer has seven rotational hard drives with 1TB of storage each. The file system and MAC files reside on these disks. The disks are used as separate devices and are not configured as a RAID array. This configuration mimics a cloud where each disk could potentially be on a separate physical machine. The operating system (Windows 7 64-bit) runs on an separate additional hard drive to avoid interfering with our experiment.

Networking. Because our file system can handle very large throughput, we used three 1Gbps cables to connect the two computers. Each computer had one network port on the motherboard and two additional network cards. After accounting for networking overhead, the 3 Gbps combined connections between the two computers can handle about 280 MB/s of data transfer as our experiments show.

Configuration. In our configuration, write operations originate from clients (simulated as threads on the Portal). Then they are processed by the Portal and multiplexed over the three network connections. Finally, data reaches the Cloud computer and is written to the appropriate disk. Reads are similarly processed, but the data flow is in the opposite direction (from the Cloud machine to the Portal).

Simulated Latency. To obtain more realistic results, we deliberately simulated 20ms round-trip time (RTT) latency between the clients and Portal, and 100ms RTT latency between the Portal and Cloud. This setting aims to resemble the scenario where the clients and Portal are both part of the same corporate network and the Cloud is a data center located elsewhere on the same continent.

7.1 Workloads

To evaluate Iris, we used the following workloads. Each workload was recorded as a trace and played back exactly under different parameterizations of our system.

- **Tar/Untar** (directory structure): Benchmarks access and modify operations on a tarball consisting of the entire Linux kernel source (420 MB, 37,000 files, and 2,300 directories).
- **IOZone** (various file access patterns): IOZone [2] benchmark of combining various operations (reread/rewrite, random read/write, backwards read, and strided read).
- **Sequential Read/Write** (throughput): Measures the performance of sequentially reading/writing ten files simultaneously, each of size 10 GB.
- **Random Read/Write** (seeks): Measures the performance of

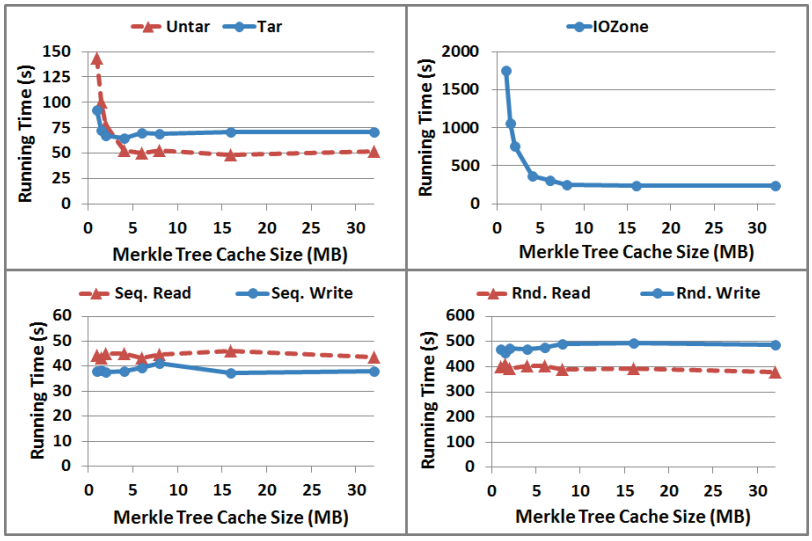


Figure 4: Workloads under different Merkle Tree Cache sizes. Time for the workload to complete vs the Merkle Tree Cache size.

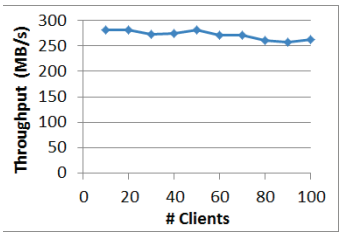


Figure 5: Avg sequential read & write speed.

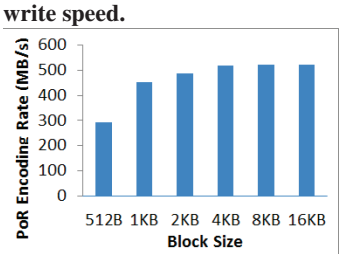


Figure 6: PoR Encoding Rate.

	Total Latency (ms)		Network I/O		Cloud Disk I/O		Portal Processing	
Portal Cache:	Hot	Cold	Hot	Cold	Hot	Cold	Hot	Cold
Create file in directory of depth 0	20.0	20.0	20.0	20.0	0.0	0.0	0.0	0.0
Create file in directory of depth 1	20.0	144.0	20.0	120.0	0.0	9.6	0.0	14.4
Create file in directory of depth 2	20.0	254.0	20.0	220.0	0.0	16.5	0.0	17.5
Create file in directory of depth 3	20.0	363.0	20.0	320.0	0.0	23.4	0.0	19.6
List directory with 10 files at depth 1	27.7	678.9	20.0	620.3	0.0	32.6	7.7	26.0
Write 1 MB file at depth 1, wait completed	24.8	138.8	20.0	120.0	0.0	0.0	4.8	18.8
Read 1 MB file at depth 1	20.0	284.2	20.0	220.0	0.0	43.7	0.0	20.5

Figure 7: Latency for different operations in Iris.

randomly reading/writing ten files simultaneously, each of size 1 GB. Reads and writes are uniformly random, and trigger seeks with almost every operation. For the random read workload, the file is first randomly written and then only the random read portion of the trace is benchmarked.

7.2 Results

Our experimental results show how Iris performs under the above workloads on the full end-to-end system described in Section 6. We note that even with seven hard drives for storage and three 1 Gbps network links between the Portal and Cloud, under no workload was the Portal the bottleneck. Depending on the workload, the limiting factor was either the network or hard drives.

Varying the Merkle Tree Cache Size: The parallel Merkle Tree Cache is crucial for the performance of our system. The cache allows the Portal to perform file operations without having to read and write entire Merkle tree paths from the server for each operation. The asynchronous cache also allows for pausing operations that are waiting to retrieve Merkle tree nodes while other operations actively use the cache.

Multiple paths can be loaded into the Merkle Tree Cache at once while maintaining consistency. In order to demonstrate the usefulness of the cache, in this experiment we varied its size (i.e., how many nodes it can hold at once) and we timed each of the workloads under different cache sizes. The results are in Figure 4.

Interpretation: As demonstrated in the figure, the Tar, Untar, and IOZone workloads greatly benefit from having a Merkle tree cache of 5 to 10 MB (about 10,000 to 20,000 nodes), whereas the sequential and random read/write workloads are mostly unaffected by the cache size.

The reason is quite simple: The Tar, Untar, and IOZone benchmarks frequently revisit the same part of the Merkle tree. For example, the Tar/Untar workloads often read/write multiple files within the same directory (and hence their Merkle tree paths share many nodes). Likewise, the random write portion of the IOZone benchmark creates a file with a large uncompactable Merkle tree which is then read sequentially and the sequential read portion of the workload yields an in-order traversal of the Merkle tree that is significantly sped up by the cache.

On the other hand, the sequential read and write workloads generate version tree nodes that are quickly compacted. Hence the Merkle Tree Cache only needs to hold a few dozen nodes at a time. The random read/write workloads are extremely intensive on the Cloud's disks. Almost every operation causes a seek, so the Cloud's disks are the bottleneck. Because the random read/write operations are executed very slowly by the Cloud's disks and the Portal parallelizes requests for the Merkle tree nodes, there is plenty of time for the Merkle tree nodes to be fetched without delaying the workload.

Scalability: In Iris, all operations are handled by the same thread pool and each file has its own queue of pending/active operations. From the Portal's perspective, there is little difference between each operation being issued by a different client and all operations being issued by the same client. Most of the overhead of having multiple clients comes from having to manage multiple TCP sockets and

their associated buffers.

We wanted then to show that Iris can easily scale to 100 clients accessing it simultaneously. To maximize both strain on the Portal's CPU and the number of cryptographic operations performed, each client generated a sequential access pattern. (With more seek-intensive access, the bottleneck would be disk seeks on the Cloud.)

We averaged the sequential read and sequential write speeds for 10 to 100 clients. Figure 5 shows the results. As can be seen, Iris consistently reads/writes at 250 MB/s to 280 MB/s. The slight performance degradation for 100 clients is due to the fact that many files are accessed at once and that causes a larger portion of disk seeks.

Latency: Figure 7 shows the latency for several basic operations in Iris. The latency is measured under two scenarios: when the portal cache is hot and cold. A hot cache means that the cache already contains all of the data (Merkle tree nodes and blocks) necessary to perform the operation on the portal alone. A cold cache means that all of the data has been evicted from the portal's cache.

The bulk of the latency (over 84%) comes from the portal-cloud and client-portal network latencies. Our results show that the latency introduced by the portal for integrity checking and cache management (denoted as portal processing time) is much smaller in comparison: less than 14% for a cold cache and less than 29% for a hot cache.

The 1 MB read operation takes about half of the time of the 1 MB write operation because the portal notifies the client that the write operation has completed while uploading the file to the cloud in the background. For the read operation, the portal must first read the file from the cloud.

The high cold cache latency for high depth operations (e.g., create depth 3 and list directory) is due to the fact that each file is represented as a separate node in the Merkle tree and tree paths are fetched one node at a time. It should be noted that this latency can be significantly reduced by having the portal fetch all nodes in a path in parallel or grouping multiple files into a single file node.

PoR Encoding Rate: Finally, we measure the rate at which the Portal can perform erasure-encoding for file system recovery if auditing detects corruption. Figure 6 shows encoding speeds for data blocks of different sizes. As can be seen, the PoR encoding rate is sufficiently fast in order to sustain a throughput of 500 MB/s for 4 KB blocks.

8. Conclusions

We have presented Iris, an authenticated file system designed to outsource enterprise-class file systems to the cloud. Iris goes beyond basic data-integrity verification to achieve two stronger properties: File freshness and retrievability. Using a lightweight, tenant-side portal as a point of aggregation, Iris efficiently processes asynchronous requests from multiple clients transparently, i.e., with no underlying file system interface changes.

Iris achieves a degree of end-to-end optimization possible only through a carefully crafted, holistic architecture, one of the systems's major contributions. Iris's architecture also relies on several technical novelties: The authenticating data-structure design and management, caching techniques, sequential-file-access optimizations, and a new erasure code enabling the first efficient dynamic PoR protocol.

Acknowledgements

We'd like to extend our thanks to Roxana Geambasu for her insightful comments on a previous draft of the paper and the anonymous reviewers for all their feedback and suggestions.

9. References

[1] Full version, `http://eprint.iacr.org/2011/585.pdf`.

[2] IOzone filesystem benchmark. www.iozone.org. 2011.

[3] `www.memcached.org`.

[4] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. Usenix, 2002.

[5] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *14th ACM CCS*, pages 598–609, 2007.

[6] M. Blaze. A cryptographic file system for Unix. In *Proc. First ACM Conference on Computer and Communication Security (CCS 1993)*, pages 9–16, 1993.

[7] K. Bowers, A. Juels, and A. Oprea. Proofs of retrievability: Theory and implementation. In *Proc. ACM Cloud Computing Security Workshop (CCSW 2009)*, 2009.

[8] G. Cattaneo, L. Catuogno, A. D. Sorbo, and P. Persiano. The design and implementation of a transparent cryptographic file system for Unix. pages 199–212, 2001.

[9] Y. Chen and R. Sion. To cloud or not to cloud? musings on costs and viability. In *ACM Symposium on Cloud Computing (SOCC)*, 2011.

[10] Y. Dodis, S. Vadhan, and D. Wichs. Proofs of retrievability via hardness amplification. In *Proc. 6th IACR TCC*, volume 5444 of *LNCS*, pages 109–127, 2009.

[11] C. Erway, A. Kupcu, C. Papamanthou, and R. Tamassia. Dynamic provable data possession. In *Proc. ACM Conference on Computer and Communications Security (CCS 2009)*, 2009.

[12] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. Sporc: Group collaboration using untrusted cloud resources. In *Proc. OSDI*, 2010.

[13] K. Fu. Group sharing and random access in cryptographic storage file systems. Master's thesis, Massachusetts Institute of Technology, 1999.

[14] K. Fu, F. Kaashoek, and D. Mazieres. Fast and secure distributed read-only file system. *ACM Transactions on Computer Systems*, 20:1–24, 2002.

[15] R. Geambasu, J. P. John, S. D. Gribble, T. Kohno, and H. M. Levy. Keypad: An auditing file system for theft-prone devices. In *Proc. European Conference on Computer Systems (EuroSys)*, 2011.

[16] E. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: Securing remote untrusted storage. In *Proc. Network and Distributed Systems Security Symposium (NDSS 2003)*, pages 131–145, 2003.

[17] M. T. Goodrich, C. Papamanthou, R. Tamassia, and N. Triandopoulos. Athos: Efficient authentication of outsourced file systems. In *Proc. Information Security Conference 2008*, 2008.

[18] A. Juels and B. Kaliski. PORs: Proofs of retrievability for large files. In *Proc. ACM Conference on Computer and Communications Security (CCS 2007)*, pages 584–597, 2007.

[19] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *Proc. 2nd USENIX Conference on File and Storage Technologies (FAST)*, 2003.

[20] S. Kamara, C. Papamanthou, and T. Roeder. Cs2: A searchable cryptographic cloud storage system. Technical Report MSR-TR-2011-58, Microsoft, 2011.

[21] J. Li, M. Krohn, D. Mazieres, and D. Shasha. Secure untrusted data repository. In *Proc. 6th Symposium on Operating System Design and Implementation (OSDI)*, pages 121–136. Usenix, 2004.

[22] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. In *Proc. OSDI*, 2010.

[23] E. Miller, D. Long, W. Freeman, and B. Reed. Strong security for distributed file systems. In *Proc. 1st USENIX Conference on File and Storage Technologies (FAST)*, 2002.

[24] A. Oprea and M. K. Reiter. Integrity checking in cryptographic file systems with constant trusted storage. In *Proc. Usenix Security Symposium 2007*, 2007.

[25] R. Pletka and C. Cachin. Cryptographic security for a high-performance distributed file system. In *Proc. 24th IEEE Conf. on Mass Storage Systems and Technologies (MSST 2007)*, 2007.

[26] R. A. Popa, J. Lorch, D. Molnar, H. J. Wang, and L. Zhuang. Enabling security in cloud storage SLAs with CloudProof. In *Proc. 2011 USENIX Annual Technical Conference (USENIX)*, 2011.

[27] H. Shacham and B. Waters. Compact proofs of retrievability. In *Proc. ASIACRYPT*, volume 5350 of *LNCS*, pages 90–107, 2008.

[28] A. Shraer, C. Cachin, A. Cidon, I. Keidar, Y. Michalevsky, and D. Shaket. Venus: Verification for untrusted cloud storage. In *Proc. Workshop on Cloud Computing Security.*, 2010.

[29] C. A. Stein, J. H. Howard, and M. Selzer. Unifying file system protection. In *Proc. USENIX Annual Technical Conference*, 2001.

[30] Q. Wang, C. Wang, J. Li, K. Ren, and W. Lou. Enabling public verifiability and data dynamics for storage security in cloud computing. In *Proc. 14th European Symposium on Research in Computer Security (ESORICS 2009)*, 2009.

[31] Q. Zheng and S. Xu. Fair and dynamic proofs of retrievability. In *Proc. 1st ACMM Conference on Data and Application Security and Privacy (CODASPY)*, 2011.

Malicious PDF Detection using Metadata and Structural Features

Charles Smutz
Center for Secure Information Systems
George Mason University, Fairfax, VA 22030
csmutz@gmu.edu

Angelos Stavrou
Center for Secure Information Systems
George Mason University, Fairfax, VA 22030
astavrou@gmu.edu

ABSTRACT

Owed to their versatile functionality and widespread adoption, PDF documents have become a popular avenue for user exploitation ranging from large-scale phishing attacks to targeted attacks. In this paper, we present a framework for robust detection of malicious documents through machine learning. Our approach is based on features extracted from document metadata and structure. Using real-world datasets, we demonstrate the the adequacy of these document properties for malware detection and the durability of these features across new malware variants. Our analysis shows that the Random Forests classification method, an ensemble classifier that randomly selects features for each individual classification tree, yields the best detection rates, even on previously unseen malware.

Indeed, using multiple datasets containing an aggregate of over 5,000 unique malicious documents and over 100,000 benign ones, our classification rates remain well above 99% while maintaining low false positives of 0.2% or less for different classification parameters and experimental scenarios. Moreover, the classifier has the ability to detect documents crafted for targeted attacks and separate them from broadly distributed malicious PDF documents. Remarkably, we also discovered that by artificially reducing the influence of the top features in the classifier, we can still achieve a high rate of detection in an adversarial setting where the attacker is aware of both the top features utilized in the classifier and our normality model. Thus, the classifier is resilient against mimicry attacks even with knowledge of the document features, classification method, and training set.

1. INTRODUCTION

To achieve higher rate of infection through social engineering and, in some cases, to avoid detection, malicious code is often embedded or packaged within documents that appear legitimate including government forms and bank statements. These maliciously crafted documents are also called trojan documents because they carry a malicious payload in a seemingly desirable document which serves as distribution mechanism for the malware.

The use of documents as a vehicle for exploitation has become more popular as the number of vulnerabilities in client applications, including document viewers, has increased [7, 8]. Indeed, the risk of infection appears to be higher because attackers entice users to open a malware-bearing document through social engineering. Moreover, the complexity and structure of modern documents can make detection of the malicious code extremely difficult: the document offers many places where code can be confused for data and many more additional layers of obfuscation compared to a Portable Executable (PE). To make matters worse, client applications, such as document viewers and data containers, are usually the consumers of foreign code and data, making them perfect targets for exploitation.

Many recent studies have demonstrated that malicious documents are frequently used in highly socially engineered phishing attacks perpetrated by groups of highly sophisticated, persistent, and targeted attackers whose goal is espionage [23, 2]. The use of trojan PDFs in targeted attacks, including those perpetrated by a class of attacker called the Advanced Persistent Threat (APT), adds urgency to countering this form of malware delivery. PDF documents have become one of the most popular file formats exploited in targeted attacks [10] and new vulnerabilities continue to be used by targeted attackers [1].

The exact mechanism of delivery and exploitation employed varies widely: in some cases, the document is used merely to exploit a vulnerability in the reader application with the document itself providing little value in terms of social engineering. For instance, some classes of web based attacks, such as those leveraging cross-site scripting, operate in this way.

In phishing attacks, the attached document augments the social engineering aspect of the attack. In some attacks, the document contains the complete malware payload the attacker wishes to deploy, while in others, the document only has enough code to download additional malware components [19, 25, 9, 27]. Although exploitation of the reader program can result in the viewer program hanging or crashing, potentially alerting the user of a problem, sometimes such faults remain hidden from the end-user because the reader program is used as plug-in in a larger program, such as Internet browsers [11, 5, 17, 3].

In highly targeted attacks, the exploit often involves opening a benign document that is extracted from the trojan document to mask the exploitation and enhance social engineer-

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

ing. Many malicious documents seem to be truly Spartan, in that they contain only malicious content without superfluous metadata or structural elements. However, some malicious documents contain extensive non-malicious metadata items and structural elements seemingly indicative of the malware author either beginning construction with an existing benign document or the author intentionally inserting non-malicious content to avoid detection. This study proposes an alternate mechanism for detecting malicious documents which seeks to leverage some of the restrictions the use of documents enforces to malware delivery.

There exist many approaches for detecting malicious documents. Signature matching is widely employed and is effective for detecting previously identified malware on a broad scale. Indeed, many signature matching systems including commodity antivirus scanners [13, 29] provide functionality specific to PDF documents so that they can be decoded to reveal malicious content and exploitation of document specific vulnerabilities. A common alternative to signature matching is dynamic analysis of documents where the behavior of the entire system or the set of programs are observed while the document is opened.

In this paper, we explore the limits of static analysis detection mechanisms that utilize machine learning techniques on document-specific attributes to identify embedded malware. Our approach addresses some of the shortcomings of existing techniques through the use of a broadly applicable mechanism to classify and characterize documents. The proposed method is vulnerability and exploit agnostic and thus seeks to remove the dependence on a priori knowledge of specific malware families and vulnerabilities while remaining computationally and operationally tractable. Moreover, we aim to further classify whether malicious documents are associated with broad based, opportunistic threats or highly targeted attacks.

As part of our analysis, we show that while the use of documents as an exploitation vector can be an enabling mechanism for the attacker, it also provides additional detection opportunities. All of the data closely associated with malicious activity can be used to aid in detection, regardless of whether the data utilized for detection is inherently malicious or not. Indeed, in signature matching systems, signatures are often generated for byte sequences highly specific to known malware families, even if those byte sequences are not malicious in and of themselves. The underlying premise and intuition of our study is that malicious documents do have similarities to other malicious documents; they also have dissimilarities to benign documents, regardless of the specific vulnerability exploited or the specific malware embedded in the document. We posit that features based on document structure and metadata are adequate for reliable document classification given appropriate statistical methods are applied to these features. At a very high level, this is similar to the semantic operation of SPAM filters that use features from the email and a machine learning engine to reliably classify email into SPAM or legitimate.

For the purpose of our analysis, documents are represented by their respective feature vectors extracted from statically processing the document files. These features are constrained to items derived from the document metadata, such as the number of characters in the title, or features derived from the document structure, such as the size of the images in the document. We show that a large number of these features are suitable for high accuracy classification. This study uses 10,000 documents from a widely available data set and 100,000 documents collected from a university network. We employed different machine learning techniques and our results show that the Random Forests algorithm performs the best with True Positives (TP) more than 0.99 (99%) while maintaining a False Positive (FP) rate of 0.002 (0.2%) or less depending on the scenario, dataset, and thresholds used. This ensemble classifier is able to classify previously unseen variants. An important finding of our study is the effectiveness of mimicry type attacks can be mitigated. This due to the large set of features we extracted and the ability to construct an ensemble classifier providing robust detection even when the top features on our training set are used as part of a mimicry attack by the attacker. By perturbating the training data to mitigate the reliance on features unevenly favored by the classifier, evasion becomes much more difficult.

2. RELATED WORK

Malware bearing documents have been the subject of much research over the years. In terms of static analysis, Li et al. [15] and Tabish et al. [28] demonstrated that detecting malicious Word documents through static analysis using n-grams representation for the document data and course dynamic analysis shows promise but also comes with limitations due to the size of the malcode. While the techniques and file format differ, the end goals are very similar to our work. Signature analysis and pattern matching has been studied extensively [26, 24] including specifically for PDF vulnerabilities [20, 16].

Addressing detection of malware-laden PDFs using static analysis, Laskov and Šrndić[14] extract, process, and classify javascript embedded in PDFs using support vector machines. However, their approach yields much lower detection rates and they do not provide a robustness analysis for their method. Munson and Cross [6] use an instrumented reader application and dynamic analysis to extract structural features of PDF documents to be used in a machine learning based classifier. While the high-level approach is similar to what is presented here, they were unable to demonstrate strong detection rates. Moreover, the framework presented here differs in that static analysis is used to extract a much larger number of features, including metadata, without seeking to fully decode the document through dynamic analysis. In both of the aforementioned studies, an important factor identified was the ability to effectively parse PDF documents which is extremely difficult, especially in the case of malicious documents. The research presented here skirts these issues by focusing on different features and different methods of extracting those features.

Furthermore, Tzermias et al. [29] showed that the effectiveness of existing antivirus systems against malicious PDF files is quite modest. To boost the detection, they used the combination of static and dynamic analysis to identify malicious documents with focus on specific vulnerabilities and yielding mixed results due to the need of VM support. Our approach is agnostic to vulnerabilities and does not require any form of execution (i.e. dynamic analysis).

Although different in terms of purpose and target, the use of statistical learning to identify malicious documents can be compared to SPAM detection, a topic on which there is a large corpus of previous work. Documents are similar

Table 1: Data Set Summary

	Training	Testing/Operational
benign (ben)	5,000	99,703
opportunistic (opp)	4,802	286
targeted (tar)	198	11
total	10,000	100,000

to email messages in that both are primarily designed for transfer of human readable text and detection is usually concerned with identifying malicious activity very early in the attack life cycle. Statistical methods, such as Bayes classification of body text, have been used effectively for years [22]. More recently, progress has been made towards effective network and transport layer identification of SPAM [4, 12]. One could consider features such as parameters of network traffic used for SPAM delivery to be analogous to the features of documents used in this study. These features are not measuring inherently malicious attributes, but they often reflect malicious activity and are useful in practice.

3. DATA & FEATURE DESCRIPTION

3.1 Data Types

This study focuses on classifying malicious documents, specifically PDF documents. The PDF document type was chosen because of it's ubiquitous use, availability of public data sets, the large number of recent vulnerabilities in the Adobe PDF reader, and the frequent use of PDF documents in targeted attacks.

In our experiments, PDF documents are classified as either benign or malicious, with malicious being further split into two categories: opportunistic and targeted. These are abbreviated "ben", "mal", "opp", and "tar" respectively. To be classified as malicious, documents must exploit a software vulnerability and execute malicious code. For the purpose of this study, documents that contain text instructing the reader to perform malicious actions (wire money), that contain hyper links to malicious content where the user must click, etc. are considered benign. Targeted attacks are separated from opportunistic ones by factors such as victim specific social engineering and correlations with other known targeted attacks.

3.2 Data Sources

There are two primary data sources used in this study. The first is the widely available Contagio data set [18] designated for signature research and testing. This source of data sets was selected because it contains a large number of labeled benign and malicious documents, including a relatively large number from targeted attacks. This source provides a few collections of documents. All of the PDF documents from "Collection 1: Email attachments from targeted attacks" were used as targeted malicious documents. The documents from "Collection 4: Web exploit pdf (I think they all are pdf) files" are used as malicious documents. The vast majority of the documents from Collection 4 were attributed to opportunistic threats and were used as such, with a few exceptions. There were 10 identical documents in collection 1 and collection 4: these were considered targeted. Also, an a few additional documents were identified as targeted through manual inspection and correlation to other targeted attacks. Lastly, "Collection 5: Non-Malicious PDF Collection" was used for benign PDF examples. A total of 10,000 documents were used from this source for the training data set.

The second collection is taken from monitoring of the network of a large university campus. These documents were extracted from HTTP and SMTP traffic. The bulk of this collection was taken from approximately 6 days of capture. Because this data is taken from a real data feed, it is termed the "operational" data set. The operational data set required labeling by the researchers to be useful for evaluation. To separate the malicious documents from the benign, a combination of 5 common virus scanners were used. The corpus was scanned with the virus scanners until signature updates ceased adding detections. The virus scanner detections continued to improve until approximately 10 days following the end of the collection. Note that no single virus scanner detected all the malicious documents. All the documents that were flagged by a single scanner were subjected to additional virus scanning and manual analysis. Two of the twelve documents identified by a lone AV scanner as malicious were found to be benign. All of the malicious documents from the week long collection were considered opportunistic as there was no evidence to support labeling them as targeted. 11 malicious PDFs associated with targeted attacks on the same organization, but representing multiple victims and attack groups, were added to this collection. These targeted PDFs were observed on the campus network over the span of approximately 18 months and were collected in the same manner. The addition of these targeted attacks was necessary to allow the operational data set to be used to evaluate targeted attack detection. A total of 100,000 unique documents were used from the operational data set.

In both data sets, only unique documents were used. Sampling, when it occured, was random. Table 1 summarizes these data sets by displaying the number documents of each class in the two sets. Note that the training data set includes equal parts benign and malicious documents which is desirable for training. The operational data set ratio of benign to malicious is intended to mirror a typical operational environment and provide insight to detection rates and false positives in a real environment. The number of targeted documents is undesirably low but is the best that could be obtained given their scarcity. The operational and the training data set are completely independent. The training set was compiled months before the operational data set.

4. CLASSIFICATION METHODOLOGY

4.1 Feature Extraction

We implemented our own program for reliably extracting features from PDF documents due to the inability of existing tools to cope with malformed documents. Regular expressions are applied to the raw document to identify and extract data for further processing, if necessary. Many of the features can be derived from simple string matching reporting solely the location of the matches. Ex. count of font objects or average length of the stream objects where the length of each stream is measured by the difference between the location of a "stream" marker and and the next "endstream" marker. Many features required extracting specific data for further normalization or processing such as the dimensions of a box object or the number of lower case characters in

Table 2: Classifier Performance Comparison

Classifier	Error Rate	Train Time	Classify Time
Naive Bayes	27%	2 sec	95 sec
Random Forest	.19%	92 sec	1 sec
Support Vector Machine	17%	218 sec	33 sec

the title. This software functions without significant PDF structure parsing or validation which is necessary for success in dealing with malformed documents and provides strong performance. This fast and loose metadata extraction intentionally results in some inaccurate or ambiguous extracted data, but the end features are deterministic. For example, a PDF edited with incremental updates may have extracted features that report an inflated object count. Additionally, instead of trying to use the correct value in the case of multiple instances of a metadata item repeated in a document, other features such as the number of times the values differ are used. Feature extraction and classification works well even on encrypted documents because in PDF documents each object/stream is encrypted individually, leaving structure and metadata to be extracted the same as normal documents.

The majority of the metadata items are inherently numeric. The other features are all extracted or transformed to make them numeric. There are largely no differences in how binary, discrete, and continuous data is handled, although some of each type exist.

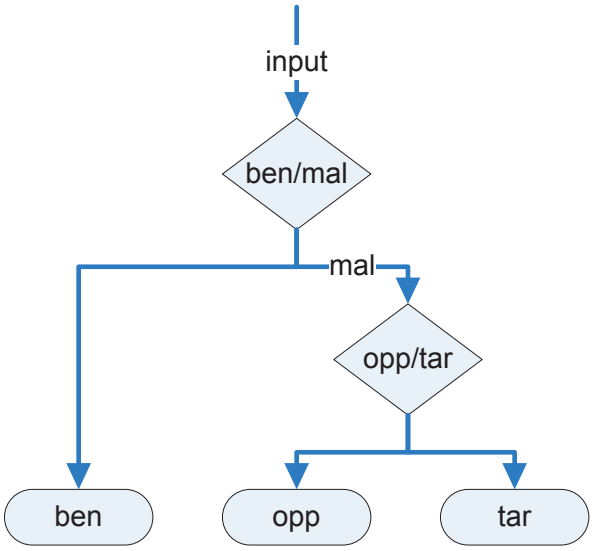


Figure 1: Dual Classifier Arrangement

4.2 Feature Selection

We selected a large number of features to characterize PDF documents. Our aim was to provide strong classification quality, including the ability to reliably distinguish targeted attacks from opportunistic attacks. In addition, the approach taken here seeks to be resilient to differences in threats and vulnerabilities by focusing on patterns in documents that apply broadly. Therefore, features are derived either from PDF document metadata or structure.

In our approach, the extracted features are designed to eliminate reliance on specific strings or byte sequences. For example, when dealing with data that might represent artifacts of specific actors, such as the author metadata item, abstracted features, such as the number of characters in the author field, are used. Similarly, features were intentionally avoided that are tightly related to specific vulnerabilities, but which have little general application, because including these features could result in strong classification for known attacks while yielding low detection rates for novel attacks.

The philosophy for feature identification was to generate as many features that parameterize the metadata and structure of the document as possible, without short-sighted regard for usefulness of individual features in discriminating between document classes. The features reflect properties of the metadata, such as the count of the characters in each field; objects/streams, such as the size and count of each; boxes and images, such as the size and location of each; data encoding methods, such as use of each data encoding method; and object types, such as count of encryption objects. In total, 202 features were chosen for use.

It is beyond the scope of this document to explain all 202 features, but some examples will be given. The names of the top features are shown in Figure 7. The count_font, count_javascript, and count_js features are determined by the number of instances of "/Font", "/JavaScript", and "/JS" markers. The count_stream_diff variable is the difference of the instances of "stream" and "endstream" markers. The relative position in the document of the last box marker (box objects are used in layout) is reported as pos_box_max. The sum of all the pixels in all the images in the document is named image_totalpx. producer_len is the number of characters in the producer metadata object and count_obj is the number of instances of the "obj" marker. Each document has a unique document identifier that should never be modified between revisions which is called pdfid0. pdfid0_mismatch reflects the number of unique instances of pdfid0 values in a document (which is usually always 1). These features are identified by either simple strings matches or more complex regular expressions applied to the raw document.

Most features are taken directly from observation of the document metadata or document structure, such as the number of font objects in the document. A few of the features are further refined by transformation of 1 or more elements. For example, one feature is the ratio of the number of pages to the size of the whole document.

4.3 Random Forests

To categorize observed documents, the features are extracted and run through a classifier generated from labeled training data. Random Forests was selected because of its effective classification capability, strong performance, and ease of use. Table 2 compares Random Forest classification performance to that of Support Vector Machine and Naive

Bayes using the training data set and the default parameters for each method. These results represent the average of multiple cross validation and timing runs. We performed all of our analysis using R [21], including the randomForest and e1071 packages.

The Random Forests classification method gives the result of classification based on the output of many individual classification trees, each of which votes for one of possible classes. Each decision tree is generated from a randomly selected subset of training data. Hence, random forests is an ensemble classifier using bagged training data. Each node in a tree is created by selecting a randomly selected subset of features and determining the best split at each node using the training data for that node. Furthermore, each tree is based on an independent subset of features. Lastly, during classification the votes of each tree determine the result.

The primary tunables for a random forest is the number of variable to try at each node (mtry) and the number of trees to create (ntree). By default, a random forest is constructed using the square root of the number of variables for mtry and 500 for ntree. It is recommended that these values be tuned by trying half the default and double the default in addition to the default. It was determined that 3 times the default is optimal for mtry and double the default is optimal for ntree for this application. These values, 1000 for ntree 43 for mtry, are used throughout this paper.

4.4 Classification Techniques

For classification, the first step is to produce a classifier using labeled training data. This classifier consists of a set of classification trees. After features are extracted from the document, they are run through the trees, each of which provides a vote in the reported classification. These votes are combined to determine the end classification. The initial training process is relatively computationally expensive but once the classifier is constructed, categorizing new observations is fast.

The goal of classification here is to not only classify documents as benign or malicious, but also to differentiate opportunistic from targeted attacks. As shown in Figure 1, unclassified documents are fed through a classifier which separates benign documents from malicious documents. Those found to be malicious are fed through a second classifier that differentiates opportunistic from targeted malicious documents. While random forest supports multiple class outcomes, this dual binary classifier arrangement is used because it makes it easier to understand and compare the two classification goals individually. The two binary classifiers are tuned independently and results for each are presented individually.

In order to tune the sensitivity of the classifier, the threshold of the votes from individual trees is adjusted. Typically, random forests operates by predicting the class which receives the most votes. Hence, the default cutoff value for a binary classifier is .5. This value can be adjusted, allowing sensitivity of the classifier to be adjusted so that the operator can select a desirable TP/FP ratio. All ROC curves presented here are created by adjusting this vote threshold value during prediction.

5. PERFORMANCE EVALUATION

Are structural and metadata features adequate for discriminating malicious documents from benign? If so, how many of these features are needed?

In Figure 2 we depict the classification error decrease as features are added to the benign/malicious classifier. The average and 95% confidence interval of the classification error of multiple randomly selected subsets of features are presented.

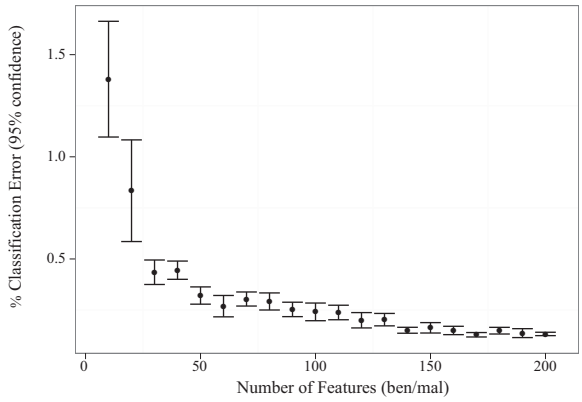


Figure 2: Error Decrease with Feature Count(ben/mal)

5.1 Classification & Detection Performance

The classifier was applied to the training data set using 10-fold cross-validation. The results from each fold are averaged to produce a single outcome for the whole set. These resulting ROC curves for the ben/mal and opp/tar classifiers are displayed in Figure 3 and Figure 4 respectively.

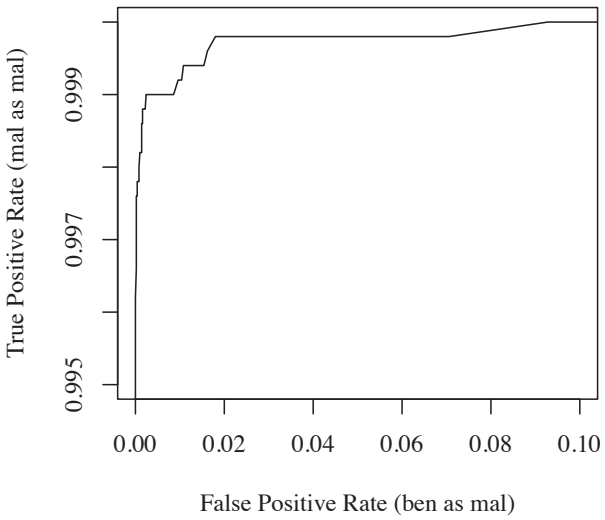


Figure 3: ROC for Training Set(ben/mal)

In addition, Table 3 and Table 4 list select data points from these graphs. The cutoff reported in these tables is the minimum percentage of votes that an observations must exceed to be considered the positive class (mal for ben/mal, tar for opp/tar).

The classifier (trained with the training set) was applied to the operational data set collected from live network observation. In lieu of presenting the ROC graphs of the classifiers applied to the operational data set, Figures 5 and 6 contain

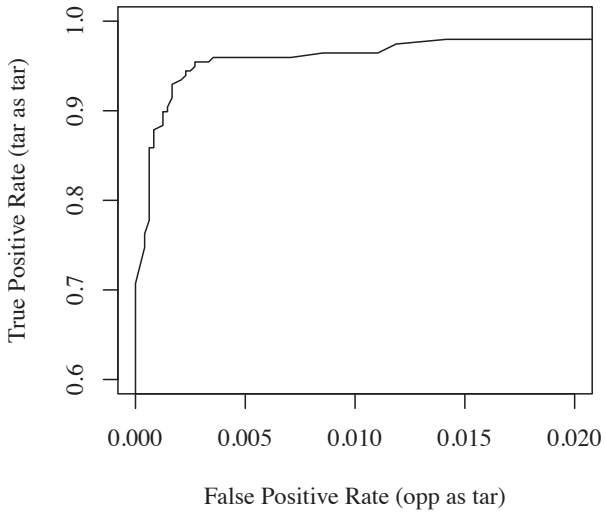


Figure 4: ROC for Training Set (opp/tar)

Table 3: FP/TP Rates: Training Set (ben/mal)

Votes	FP Rate	TP Rate	FP Count	TP Count
0.7	0	0.9942	0	4971
0.6	0	0.9962	0	4981
0.5	0.0008	0.998	4	4990
0.4	0.0014	0.9986	7	4993
0.3	0.0034	0.999	17	4995
0.2	0.0076	0.999	38	4995
0.1	0.0208	0.9998	104	4999

Table 4: FP/TP Rates: Training Set (opp/tar)

Votes	FP Rate	TP Rate	FP Count	TP Count
0.7	0.00125	0.8889	6	176
0.6	0.00167	0.9195	8	182
0.5	0.00271	0.9495	13	188
0.4	0.00333	0.9545	16	189
0.3	0.00437	0.9595	21	190
0.2	0.00583	0.9595	28	190
0.1	0.00854	0.9645	41	191

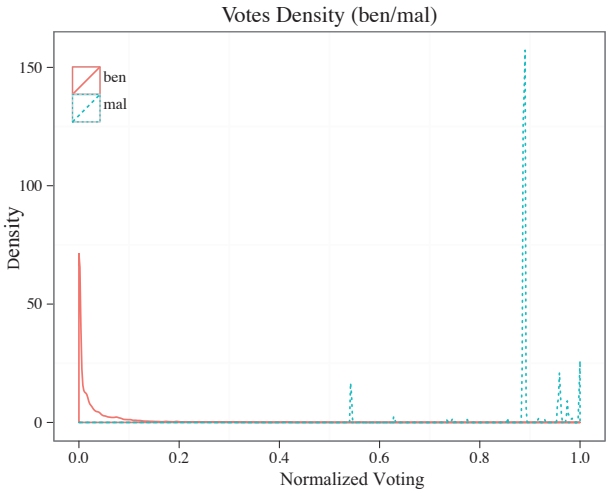


Figure 5: Classification Votes Density (ben/mal)

the density plots of the votes for the two classes in each of the binary classifiers.

These plots show the separation between the classes in each classifier, focusing on the ratio of votes that observations of a given class receive for the positive class from the the independent trees in the forest. Perfect classification would result in the negative class being a spike at 0 and the positive class being located at 1. Note that while the operational data set is largely independent of the training set, the classifiers still provide strong discrimination between the classes, even if some trees contribute an incorrect vote. These plots also clearly visualize the ability the operator has to tune the sensitivity of the classifier by adjusting the vote threshold.

In Table 5 and Table 6 we list select data points for the ROC from this data.

5.2 New Variant Detection

To demonstrate the resiliency of structural and metadata features across differing data sets, the results of the off-the-self AV scanners were used to distinguish "variants" of very similar malicious documents. The results of the 5 AV scanners were used to create a variant identifier which is the 5-tuple of the signature name of each scanner. The results of categorization of the samples into variants are shown in Table 7.

This categorization of samples into groups of variants resulted in a significant reduction in the opportunistic samples but a relatively minor reduction in the targeted samples. This is consistent with the wider and more automated distribution of opportunistic samples as well as the more exclusive distribution and higher likelihood to include manual modifications to support AV evasion of targeted samples.

To the degree possible to discern from the names, the AV signatures are related to the exploit and javascript used in exploitation. There were relatively few signatures that appeared to be related to the actual malware families concealed in the document. Indeed, many of the documents merely contain shellcode which in turn downloads specific malware. Of the two groups of targeted variants in the operational data set, one pair was attributed to the same persistent actor/embedded malware family, while the other pair of documents were from separate actors/embedded malware families. In both cases, the documents had very similar structure and metadata, leading to the conclusion they they were derived from the same document template.

There is a relatively small amount of overlap in the variants from the training and operational data sets. The ability of the classifier to effectively classify new variants that were not included in the training set demonstrates that the features used for classification are durable. Variations in malicious documents that require unique signatures can be detected with a common classifier based on metadata and structure.

5.3 Comparison to PJScan

To demonstrate the effectiveness of the mechanisms presented here and to add further validation to the quality of the data sets used, the results of using PJScan are presented

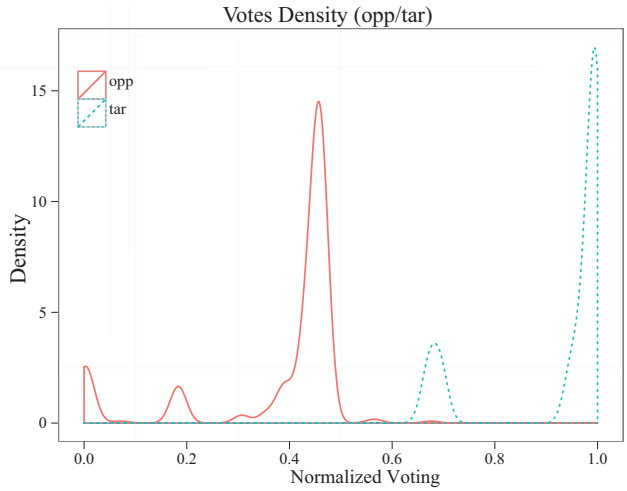


Figure 6: Classification Votes Density (opp/tar)

Table 5: FP/TP Rates: Operational Set (ben/mal)

Votes	FP Rate	TP Rate	FP Count	TP Count
0.8	0.00021	0.9327	21	277
0.7	0.00057	0.9461	57	281
0.6	0.00132	0.9529	132	283
0.5	0.00244	1.0000	243	297

here. PJScan is not designed to separate targeted from opportunistic attacks. Thus, we only used ben/mal classification to separate malicious from benign documents regardless of the type of threat. The classifier was trained on the 5,000 malicious documents in the training set with default parameters. Results are shown in Table 8. PJScan only returns a result for documents from which it can extract javascript. Therefore, in our experiments, we only count the number of documents for which a result is returned. The classification error rate is given for those documents which a result is returned.

PJScan was unable to classify many malicious documents. Manual analysis reveals that many of the malicious documents PJScan can't analyze do have javascript in them but have the javascript in atypical locations. Javascript code inside document metadata sections or inside corrupted document structures are not correctly parsed by PJScan, which is a known limitation. It appears that the newer operational data set has a much higher prevalence of these conditions which prevent successful analysis. PJScan provides decent results for the training set, but when the trained classifier is applied to the operational data set, the quality of classification drops dramatically.

The biggest limitation of PJScan applied to the data sets in this study is the inability to successfully extract the features used for classification. The mechanisms presented here, which utilize simple signature matching without document parsing or decoding, compare favorably in this as they can be more reliably extracted in practice. The mechanisms presented here also compare favorably when considering the adequacy and durability of the classifier when applied to the training data and extrapolated to other, independent data sets.

Table 6: FP/TP Rates: Operational Set (opp/tar)

Votes	FP Rate	TP Rate	FP Count	TP Count
0.8	0.0000	0.82	0	9
0.7	0.0000	0.82	0	9
0.6	0.0035	1.00	1	11
0.5	0.0105	1.00	3	11

Table 7: Variants in Data Sets

	Training	Operational	Overlap
opp samples	4,802	286	-
opp variants	812	31	6
tar samples	198	11	-
tar variants	186	9	1

5.4 Computational Complexity

The document classification process can be divided into three logical steps: feature extraction, classifier training, and classification of new observations. Feature extraction must occur for both training data and new data to be classified. The majority of the processing in feature extraction is dedicated to matching signatures on the documents. This facet was poorly optimized in the implementation used for this study where multiple signatures were applied to the document serially, with each of the signatures requiring another pass through the document. This implementation could be improved by making this signature matching parallel, which would improve performance about an order of magnitude and put performance roughly on par with conventional antivirus scanners.

Once the features are extracted from documents to be classified, running these observations through the classifier is extremely fast. Training the classifier is more expensive, but this only needs to occur infrequently. Table 9 demonstrates the run times of these operations applied to the training data set, which contains 10,000 documents. The experiments were performed on an Intel Xeon X5550 processor running 2.67GHz CPU. All the applications were executed in single-thread mode.

Table 9: Run Times on Training Data

Operation	Time
Feature Extraction	14 min
Classifier Training	38 sec
Observation Classification	1 sec

Similarly, little effort was placed into minimizing use of memory. However, for all operations, memory usage was negligible except for training the classifier, which required about 1 GB of RAM.

6. ADVERSARIAL ANALYSIS

It is important that any detection mechanism demonstrate resistance to intentional evasion. Therefore, the robustness of the selected features under mimicry and evasion attacks is crucial to the actual detection rates that can be achieved in a real-world environment. The detection mechanism presented in this paper is designed to classify documents based on similarity to past documents of the same class. These

Table 8: PJScan results

Data Set	Class	Classified	Not Classified	Classification Error	Total Detection Rate
Training	ben	5%	95%	1%	-
Training	mal	85%	15%	9%	78%
Operational	ben	3%	97%	1%	-
Operational	mal	17%	83%	36%	3%

similarities between documents can arise from a wide spectrum of root causes varying among necessity, convenience, convention, and ambivalence. Presumably some of the attributes of malicious docs are easy to modify and others are more difficult. For example, while use of javascript is often not strictly required for exploiting vulnerabilities in PDF readers, it is often the most practical method for triggering a successful exploit. Hence the features related to the existence of javascript may be hard for attackers to modify. Alternatively, it may be trivial to spoof or remove metadata such as the producer field. It is infeasible to fully enumerate or to accurately predict or anticipate all the methods used to attempt evasion, especially as some of the factors are dependent on the attacker and attack vector. Some constraints on evasion are also caused by the use of this mechanism in parallel to other techniques.

Note that the mere existence of benign elements or lack of specific malicious artifacts is not sufficient to evade detection. Some of the malicious documents evaluated in this study contained large portions of benign content indicative that the malware packager either intentionally added benign elements to a malicious document or that the exploit and malware were added to an existing benign document. Conversely, many malicious documents are devoid of optional metadata. The ability to correctly classify documents despite conflicting alignment or differences between documents demonstrates the value of the machine learner.

The use of a training set from a different organization that was was published months before the use of the classifier provides strong indication that at least some resiliency exists in the similarities used as the basis for detection in this study. In the next part of this section, we will provide experimental evidence that shows the robustness of our approach on attacks that attempt direct evasion.

6.1 Mimicry Attack Effectiveness

One likely evasion technique would be to mount a mimicry attack where malicious documents are purposefully modified to "normalize" some of their features and make them similar to benign documents while still retaining the embedded malicious content. If the attacker has knowledge of specific features used in the classifier and their importance, along with a good representation of what the defender considers as normal, the attacker can focus on mimicking the features most important for classification.

To simulate mimicry of document properties, the authors modified the top ranked features of malicious observations and subjected these modified observations to the classifier. For simplicity's sake, the documents themselves are not actually modified, but rather the previously extracted feature sets are modified. Specifically, the mean and standard deviation of the benign observations is calculated and the values for the malicious documents are replaced with random values which fit a normal distribution with the same mean and standard deviation. Note that this method may result in doctored features that are inconsistent or illogical. The six most important features, as ranked by the mean decrease in accuracy measurement, were selected for evasion testing. These features are ranked above the others with some amount of separation, as shown in Figure 7.

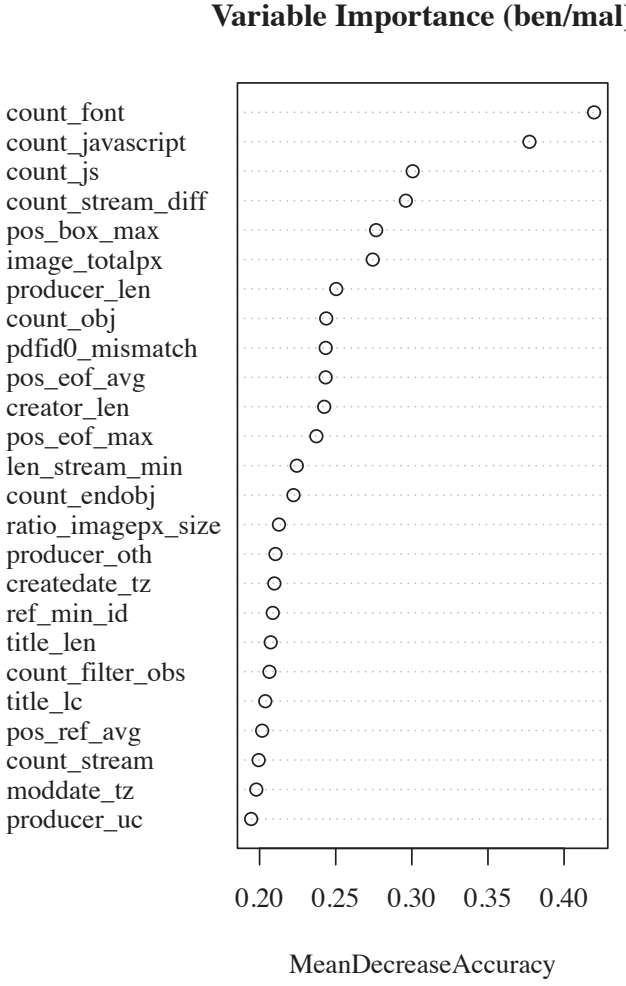


Figure 7: Feature/Variable Importance (ben/mal)

By causing the malicious samples to mirror the top six features of the benign, the benign-malicious classifier error rate can be raised a great degree, as shown in Table 10. The average of the results of 5 independent trials using 10-fold cross validation is presented.

By manipulating the most heavily used or distinctive features, it is possible to severely curtail the detection capabilities of the classifier.

Table 10: Mimicry: Classifier Error Increase

Features Mimicked	Classification Error (%)
None	0.14
count_font	12.26
(+) count_javascript	17.04
(+) count_stream_diff	20.01
(+) count_js	20.07
(+) pos_box_max	22.18
(+) image_totalpx	22.30

6.2 Countering Mimicry

The best reaction to changes in document attributes leading to mis-classification is to retrain the classifier, causing the classifier to adjust how it treats the mimicked features. If retraining the classifier it is not adequate to raise classification rates to an acceptable level, additional features can be discovered and utilized instead. This tactic is reactionary at best and cannot ensure detection of documents that are very dissimilar to historical examples of documents of the same class. To be able to detect intentional evasion, proactive measures must be taken.

An obvious reaction to mimicry attacks on the features heavily employed by the classifier is to remove them altogether and rely on the other features. An important distinction is that variable importance, as reported by random forests, is an indication of the value of the feature as used in the classifier. However, that a feature has a high importance does not necessarily mean that the feature is useful for classification on it's own nor does it mean that the classifier has to rely heavily on that feature for successful classification. Table 11 shows the increase in classification error as the top features are removed.

Table 11: Classifier Error with Features Removed

Features Removed	Classification Error (%)
None	.14
count_font	.21
(+) count_javascript	.28
(+) count_stream_diff	.28
(+) count_js	.29
(+) pos_box_max	.29
(+) image_totalpx	.29

Removing the top ranked features has a surprisingly low affect on classification error because so many other useful features are retained. If the attacker is able to only modify a few attributes of malicious documents, and the defender is able to anticipate these, removing features may be an acceptable counter-measure. It is desirable to be able to counter evasion without fully negating the predictive value of variables targeted for evasion. One method of achieving this result is to vary (perturbate) the training set such that the resulting classifier is no longer as susceptible to evasion. The perturbation is performed by artificially modifying the features of a subset of the malicious observations in the training set to increase the variance of these features thus making them less "normal". The loss of a focal point due to the increased variance reduces the importance of these features without fully eliminating them.

To test the effectiveness of perturbation, the same method used to simulate evasion is used to modify a subset of the observations in the training set. The top six features of a subset of the malicious observations is set to values taken from a randomly generated normal distribution mirroring the mean and standard deviation of the benign observations. Table 12 shows the results of testing using the perturbation method. The average of the results of 5 independent trials using 10-fold cross validation is presented. The training data is perturbated and the resulting classifier is used both on the remaining unmodified training data and the same training data modified to simulate mimicry evasion. The percentage of the training data perturbated is varied, demonstrating a trade-off between accuracy with historical data and evasion resistance.

Table 12: Classification Error with Training Data Perturbation

% Perturbation	Original Data	Mimicry Data
0	.14	26.12
.05	.14	14.11
.1	.15	9.19
.5	.15	1.80
1	.16	1.13
5	.21	.69
10	.22	.52
50	.26	.16
100	2.06	.12

7. FUTURE WORK

As an extension to the presented results, It would be valuable to determine how well the same detection and classification techniques apply to other documents types. Another potential research avenue would be to determine how well the features used for classifying documents are suited to grouping malicious documents by malware family. Another option would be to combine the features used in this study with other features, such as features derived from content analysis of the document, those derived from transport of the document over the network, recipient oriented features, and attacker oriented features. Lastly, the performance of this mechanism could be systematically compared to other techniques.

8. CONCLUSION

We presented a classification approach for PDF documents that have embedded malicious code. We showed that by extracting a wide-ranging feature set we can create a robust malware detector and classifier that yields very high rates of true positives (TP) while maintaining a low rate of false positives (FP). We evaluated different machine learning techniques and we show that Random Forests appears to be the most effective.

The experimental results obtained using more than 5,000 malicious documents and 100,000 benign ones yield classification rates above 99% while maintaining low false positive rates of 0.2% or less. We can also achieve classification of the malware documents into opportunistic and targeted categories. We also demonstrate that the classifier is effective at identifying new variants by applying the classifier trained on one data set to a totally independent training set with

great success.

Furthermore, our approach is robust against direct evasion and mimicry attacks that target the top classification features. We achieve that by artificially varying the ranges of the top features effectively reducing their influence on the detection. The result is that classification depends more equally on a very large number of features, making evasion much more difficult to accomplish. Our experiments show that this strategy is still effective in detecting and classifying malware documents allowing for a defender to create a classifier that exposes malware that attempts to mimic the normality of the documents in the training set.

9. REFERENCES

[1] Adobe Inc. Adobe security advisories: APSA11-04 - security advisory for adobe reader and acrobat. http://www.adobe.com/support/security/advisories/apsa11-04.html.

[2] D. Alperovitch and M. (Firm). Revealed operation shady RAT. http://www.mcafee.com/us/resources/white-papers/wp-operation-shady-rat.pdf, 2011.

[3] C. Andrianakis, P. Seymer, and A. Stavrou. Scalable web object inspection and malfease collection. In *Proceedings of the 5th USENIX conference on Hot topics in security*, pages 1–16. USENIX Association, 2010.

[4] R. Beverly and K. Sollins. Exploiting transport-level characteristics of spam. In *Conference on Email and Anti-Spam*, 2008.

[5] M. Cova, C. Kruegel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proceedings of the 19th international conference on World wide web*, pages 281–290. ACM, 2010.

[6] J. S. Cross and M. A. Munson. Deep PDF parsing to extract features for detecting embedded malware. Technical Report SAND2011-7982, Sandia National Laboratories, Sept. 2011.

[7] R. Dhamankar, M. Dausin, M. Eisenbarth, and J. King. The top cyber security risks. http://www.sans.org/top-cyber-security-risks/, Sept. 2009.

[8] J. Drake. Exploiting memory corruption vulnerabilities in the java runtime. 2011.

[9] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 88–106, 2009.

[10] F-Secure. PDF most common file type in targeted attacks - F-Secure weblog : News from the lab. http://www.f-secure.com/weblog/archives/00001676.html.

[11] C. Grier, S. King, and D. Wallach. How i learned to stop worrying and love plugins. In *In Web 2.0 Security and Privacy*. Citeseer, 2009.

[12] G. Kakavelakis, R. Beverly, J. Young, T. Limoncelli, and D. Hughes. Auto-learning of SMTP TCP Transport-Layer features for spam and abusive message detection. In *LISA 2011, 25th Large Installation System Administration Conference*, Boston, MA, USA, Dec. 2011. USENIX Association.

[13] T. Kojm. ClamAV releases. http://lurker.clamav.net/message/20100816.145508.07ae1603.en.html, Aug. 2010.

[14] P. Laskov and N. Šrndić. Static detection of malicious javascript-bearing pdf documents. In *Annual Computer Security Applications Conference*, 2011.

[15] W.-J. Li, S. J. Stolfo, A. Stavrou, E. Androulaki, and A. D. Keromytis. A study of malcode-bearing documents. In B. M. Hämmerli and R. Sommer, editors, *DIMVA*, volume 4579 of *Lecture Notes in Computer Science*, pages 231–250. Springer, 2007.

[16] D. Maiorca, G. Giacinto, and I. Corona. A pattern recognition system for malicious pdf files detection. In P. Perner, editor, *Machine Learning and Data Mining in Pattern Recognition*, volume 7376 of *Lecture Notes in Computer Science*, pages 510–524. Springer Berlin / Heidelberg, 2012.

[17] A. Niki. *Drive-by download attacks: Effects and detection methods*. PhD thesis, Master's thesis, Royal Holloway University of London, 2009.

[18] M. Parkour. contagio: Version 4 april 2011 - 11,355+ malicious documents - archive for signature testing and research. http://contagiodump.blogspot.com/2010/08/malicious-documents-archive-for.html.

[19] M. Ramilli and M. Bishop. Multi-stage delivery of malware. In *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on*, pages 91–97. IEEE, 2010.

[20] S. Rautiainen. A look at portable document format vulnerabilities. *Information Security Technical Report*, 14(1):30–33, 2009.

[21] R project. http://www.r-project.org/.

[22] M. Sahami, S. Dumais, D. Heckerman, and E. Horvitz. A bayesian approach to filtering junk E-Mail. *AAAI 98 Workshop on Text Categorization*, July 1998.

[23] Shadows in the cloud: Investigating cyber espionage 2.0. http://shadows-in-the-cloud.net/, 2010.

[24] M. Shafiq, S. Khayam, and M. Farooq. Embedded malware detection using markov n-grams. *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 88–107, 2008.

[25] D. Stevens. Malicious pdf documents explained. *Security & Privacy, IEEE*, 9(1):80–82, 2011.

[26] S. Stolfo, K. Wang, and W. Li. Fileprint analysis for malware detection. *ACM CCS WORM*, 2005.

[27] B. Stone-Gross, M. Cova, C. Kruegel, and G. Vigna. Peering through the iframe. In *INFOCOM, 2011 Proceedings IEEE*, pages 411–415. IEEE, 2011.

[28] S. Tabish, M. Shafiq, and M. Farooq. Malware detection using statistical analysis of byte-level file content. In *Proceedings of the ACM SIGKDD Workshop on CyberSecurity and Intelligence Informatics*, pages 23–31. ACM, 2009.

[29] Z. Tzermias, G. Sykiotakis, M. Polychronakis, and E. Markatos. Combining static and dynamic analysis for the detection of malicious documents. In *Proceedings of the Fourth European Workshop on System Security*, page 4. ACM, 2011.

Jarhead
Analysis and Detection of Malicious Java Applets

Johannes Schlumberger
University of California, Santa Barbara
js@cs.ucsb.edu

Christopher Kruegel
University of California, Santa Barbara
chris@cs.ucsb.edu

Giovanni Vigna
University of California, Santa Barbara
vigna@cs.ucsb.edu

ABSTRACT

Java applets have increasingly been used as a vector to deliver drive-by download attacks that bypass the sandboxing mechanisms of the browser's Java Virtual Machine and compromise the user's environment. Unfortunately, the research community has not given to this problem the attention it deserves, and, as a consequence, the state-of-the-art approaches to the detection of malicious Java applets are based either on simple signatures or on the use of honeyclients, which are both easily evaded. Therefore, we propose a novel approach to the detection of malicious Java applets based on static code analysis. Our approach extracts a number of features from Java applets, and then uses supervised machine learning to produce a classifier. We implemented our approach in a tool, called Jarhead, and we tested its effectiveness on a large, real-world dataset. The results of the evaluation show that, given a sufficiently large training dataset, this approach is able to reliably detect both known and previously-unseen real-world malicious applets.

1. INTRODUCTION

Malicious web content is a major attack vector on the Internet [32]. Typically, the attacker's goal is to install and run a piece of malware on the victim's computer, turning it into a member of a botnet. To this end, attackers try to lure users onto malicious web pages that contain malicious code [31]. This code might trick the victim into downloading and running a malware program (through social engineering). Alternatively, the malicious code might try to exploit a vulnerable part of the victim's browser, such as an ActiveX control, the JavaScript engine, or the Java plugin. Attacks that exploit (browser or plugin) vulnerabilities when users visit malicious web pages are called drive-by download attacks. During the last two years, there has been a tremendous increase in Java-applet-based attacks - more than 300% in the first half of 2010 [15] alone. Interestingly, these exploits are widely ignored by security researchers so far [22]. This is despite the fact that commercial exploit toolkits such as "Blackhole" or "Bleeding Life" are using Java-based attacks to compromise large numbers of computers [22, 23].

A Java applet is a piece of software designed to be executed in a user's browser by a Java virtual machine (JVM [24]). Applets are embedded into web pages. They are automatically loaded and executed if the browser has a Java plugin installed (once the page has loaded), and graphical content is appropriately displayed by the browser. Applets predate cascading style sheets (CSS), and have typically been used for formatting and navigation purposes before more modern mechanisms became available (e.g., Flash, JavaScript, and HTML5). However, an applet can be any program and is not bound to a specific purpose.

Internet users are often not aware of the existence of applets and the Java plugin. Thus, they are not careful about keeping the plugin up to date, or care to disable it when visiting untrusted sites, even though multiple vulnerabilities exist and keep emerging with different versions of Java plugins [3, 4, 5, 6, 7]. As a result of this lack of awareness, Java plugins are still widely deployed, and about 85% [22] of all browsers today have the Java plugin installed and enabled. In addition, 42% of all browsers running with Java plugins have known vulnerabilities [21] and are thus susceptible to attacks [19, 22].

Traditionally, malicious content is recognized by matching programs (or data) against known malicious patterns, called *signatures* [2, 33]. Unfortunately, this approach is susceptible to obfuscation techniques that make malicious content look different and not match an existing signature [25]. Additionally, this approach can only detect already-known attacks, and new signatures need to be added to recognize new variations of an exploit.

Another approach to detect malicious web content is to use low interaction honeyclients [10, 30]. Honeyclients are built as instrumented, automated browsers that visit web pages and monitor changes to the system [1, 32]. If abnormal behavior, such as creation of files or processes, is detected, the page is deemed malicious. This approach can only detect malicious behavior that the system was explicitly set up to detect. In particular, if the software targeted by an exploit is not running on the honeyclient, it cannot be exploited, and as a result, no malicious intent can be detected. It is a very tedious, human-intensive task to keep a honeyclient running with all the different versions and combinations of software packages to ensure that no exploits are missed. Thus, honeyclients can have high numbers of false negatives [34]. Moreover, malware authors attempt to fingerprint honeyclients, by testing for specific features in the

ACASC '12 Dec. 3-7, 2012, Orlando, Florida USA

environment it provides. When such mechanisms detect a honeyclient, the code will not behave maliciously, and therefore, evade detection.

To address the growing problem of malicious Java-applets, we have developed a system, called Jarhead, that statically analyzes Java applets. This system uses a set of features and machine learning techniques [17] to build a classifier that can decide whether an applet is malicious or benign. Our approach works independently of the attacked Java version and without recreating a realistic runtime environment. It allows us to analyze applets fast and efficiently and to categorize them as benign or malicious.

We have tested our system on real world data sets with more than 3,300 applets (collected both manually and from the public submission interface of Wepawet, a popular and publicly-available system for the analysis of web threats [8]). Our evaluation shows that Jarhead works with high accuracy and produces very low false positives.

The main contributions in this paper are:

- We address the problem of malicious Java applets, a problem on the rise that is currently not well addressed by existing work. To this end, we have developed a reliable detector for malicious Java applets, which we call Jarhead.
- Jarhead uses static analysis and machine learning techniques to identify malicious Java applets with high accuracy. Our detection is fast and robust to obfuscation. It also requires little maintenance compared to signature-based detection systems and honeyclients; we do not need to collect signatures or maintain a realistic and complete runtime environment.
- We executed Jarhead on a large collection of malicious Java applets collected in the wild. We found that our system detected malicious programs with high accuracy, and we were able to identify a new (zero-day) exploit against a previously unknown vulnerability.

2. BACKGROUND

Before we describe our system, this section provides some brief background on applets and the Java sandboxing mechanism [16].

Applets usually come packaged as archives (called Jar files), which consist of individual class files containing Java bytecode (for brevity's sake, we refer to both individual class files and archives of class files as Jar files or Jars throughout the paper). In addition to the Java class files, a Jar archive can contain arbitrary additional files. If the Jar contains an applet, it has to hold at least the files containing the applet's code. Additional contents typically include a manifest file, describing the starting point for execution of this applet, version information, or other package-related data in name-value format. Additionally, Jars often contain additional data needed by the applet, such as media files or copyright information.

To protect against malicious applets, the JVM contains (sandboxes) an applet and heavily restricts its permissible operations when it is not loaded from disk. Applets loaded over the network can, for example, only access resources on the remote host they were loaded from, and they cannot read certain system properties on the client side (such as usernames or the current working directory). These protection mechanisms are part of the same-origin policy [9] of the browser, designed to prevent untrusted sites from interfering with the user's communication to trusted sites. Moreover, a sandboxed applet cannot access client resources, such as the file system. By restricting the abilities of untrusted mobile code, its abilities to infect an end user's machine or to tamper with her data are severely limited. Furthermore, by preventing the applet from loading native code that is not verifiable by the JVM, creating a class loader, or changing the Java security manager, the JVM provides a safe execution environment. In the absence of bugs in the described sandbox implementation, this enables end users to safely browse the web even in the presence of sites that serve malicious Java code.

Applets that are digitally signed with a certificate (that is, certificates trusted by the user) run effectively outside the sandbox. In such cases, the previously-described restrictions do not apply. The browser, encountering an applet with a signature, will usually display a dialog window asking the user if he trusts the applet's certificate or not. If the user accepts the certificate, the applet runs with full privileges, otherwise, it is sandboxed. An applet that is started from JavaScript remains always sandboxed.

Malicious applets try to escape the sandbox and install malware on a victim's computer. To achieve this, some applets try to trick careless users into trusting their certificates. Others target the JVM itself by trying to exploit a vulnerability in the Java plugin [3, 5, 6, 7], effectively disabling the sandbox and turning the applet into a full-fledged, non-restricted program with permissions equal to that of the user running the browser.

In this paper, we address the problem of malicious Java applets trying to exploit the Java virtual machine. Even though Jarhead does not defend against social engineering techniques, our system is able to identify applets used as a necessary part of these attacks as malicious.

3. JARHEAD SYSTEM OVERVIEW

This section introduces our analysis system, called Jarhead. We will describe design choices and provide a high-level overview of its operation.

Jarhead relies on static analysis and machine learning to detect malicious Java applets. Jarhead operates at the Java bytecode level. That is, Jarhead analyzes the bytecode that is part of a class file. Java bytecode is the encoding of the machine instructions for the JVM, similar to binary code for "bare metal" machines. In addition to the bytecode, we collect strings from the constant pool. The constant pool holds strings and other data values, and it is also part of a Java class file. Usually, there is one class file per class of the program. Java class files can be seen as the equivalent of binaries in a specific format (such as ELF on Linux or PE on Windows).

To analyze a Jar file, we extract its contents and then disassemble *all* class files contained in the Jar into a *single* (disassembled) file. Furthermore, we collect certain statistics about the applet: The number and type of files contained, its total size, and the total number of class files. We pass these statistics and the disassembled file to our feature extraction tool, which derives concrete values for a total of 42 features. Our full set of features will be discussed in more detail in the next section.

Jarhead operates in two steps: First, we use a training set of applets - each applet known to be benign or malicious - to train a classifier based on the features Jarhead extracts from applets. After the training phase, we use this classifier in a second step to perform detection of malicious applets.

Our analysis system has to be robust to obfuscation, so that attackers cannot easily evade detection by small changes to their exploits. It should ideally also not require reconfiguration or retraining when new vulnerabilities are exploited, to avoid the window of "blindness" when a new exploit surfaces. Furthermore, the system should be fast and not require human interaction.

Jarhead's analysis focuses only on the Jar file (and the Java classes within), without looking at the surrounding web page or the interaction between the applet and its environment (and the browser). Thus, Jarhead does not require access to the parameters stored in the web page containing the applet, nor to other parts of the environment in which the Jar was deployed. This enables Jarhead to work with in-vitro samples, similar to anti-virus software. The main advantage of this approach is that we do not require parts of the applet that are sometimes hidden by the attacker through obfuscation. In fact, often, parameters are used to provide shellcode (or a key used to decipher shellcode strings) to the applet. When these parts are missing for an offline sample, or they are successfully hidden from a dynamic analysis system during online processing, detection fails. We found that, very often, malicious Java applets require input from the surrounding environment. Unfortunately, this information is often missing from samples shared between security analysts and anti-virus companies. In these cases, a dynamic analysis approach (which is often used for analyzing web and script content) simply fails. More precisely, dynamic analysis systems are unable to start or even load the code successfully and cannot gain information from a program run.

Malicious code also uses fingerprinting to evade dynamic analysis, by not displaying malicious behavior while analyzed. Additionally, dynamic analysis through emulation works at a slower speed than execution on real hardware. Therefore, a malicious program can be written in such a way that it delays dynamic analysis by executing slowly within the analysis system. When an analysis run is terminated before the malicious activity starts, the malware will be falsely deemed benign. The same program can execute quickly on real hardware, successfully infecting the users browser. By using static analysis, Jarhead is able to analyze malware samples that resist dynamic analysis systems in an efficient way, while being robust to fingerprinting.

Furthermore, many Java vulnerabilities only apply to specific versions of the JVM. To be able to execute samples, even those that are available to the analysis system together with their external input data (i.e., the aforementioned cipher keys or shellcode strings), one would need to run these samples in different versions of the JVM, using different environments. Jarhead does not suffer from this problem since its analysis is static and environment-independent.

Another limitation for dynamic analysis systems is that they can only learn about the part of the program that is executed. Usually, the code coverage for one run of a program is not complete. Additionally, due to time constraints, dynamic analysis systems often do not wait for even one full program run to finish. Jarhead does not suffer from this limitations and analyzes the entire code.

Java bytecode was specifically designed to be verifiable and easy to parse. Static analysis, therefore, works well on bytecode, and does not suffer from a lot of the limitations [28] common to other binary program formats, such as computed jumps. Furthermore, a Java program does not have the ability to dynamically generate code at runtime, which is usually a problem for static analysis techniques. Thus, even when attackers make use of code obfuscation, a static analysis approach (such as the one proposed in this paper) can obtain sufficient information to correctly classify an unknown program (as our results demonstrate).

4. FEATURE DISCUSSION

This section describes the features that we extract from a potentially-malicious Jar file and that we use in our classification process. We categorize our features and argue why they are difficult to evade.

Our features can be divided into seven categories. The first three categories address obfuscation; they form the group of the *obfuscation features*. The other four categories aim at exposing the purpose of an applet by statically examining its potential behavior. These four categories form the group of *behavioral features*.

4.1 Obfuscation Features

Obfuscation is an important aspect for all malware today. Obfuscated code differs from other code because it is generated by obfuscation kits in an automated way. These kits chop up string constants and introduce additional statements or declarations without changing the semantics of the original code. We use different code metrics as features to find differences between obfuscated and non-obfuscated code. Of course, obfuscation alone is not sufficient to identify a Java program as malicious, since obfuscation kits are also used by benign code for legitimate purposes. However, while manually examining many Java applets collected from the Internet, we found that obfuscation is overwhelmingly used by malicious applets. Furthermore we do not rely purely on the obfuscation features but our other features help in distinguishing benign obfuscated applets from malicious ones.

4.1.1 Code Metrics

We collect a number of simple metrics that look at the size of an applet, i.e., the total number of instructions and the number of lines of disassembled code, its number of classes, and the number of functions per class. In addition, we measure the cyclomatic complexity of the code [27].

Cyclomatic complexity is a complexity metric for code, computed on the control flow graph (CFG). It counts the number of linearly independent paths through the code. A code that does not have branches has cyclomatic complexity 0. If the code has exactly one branch, the cyclomatic complexity would be one; if one of the two branches contains another branch before the two paths merge, the cyclomatic complexity would be three, and so on. A measurement is computed per function, and we use the average of all measurements as one feature.

To find semantically useless code, we measure the number of dead local variables and the number of unused methods and functions. For our analysis, a variable is dead if the

variable is declared and defined at least once, but its value is never read (loaded) after the definition. A method or function is unused if it is defined within the applet, but never invoked by any code within the applet.

The code metric features measure either the size of the applet or the quality of the code. Changing the size of the applet will either remove parts of the applet that were specifically added to guard against signature-based detection or will increase the size of the applet, and hence, increasing the distribution cost (bandwidth, time) for campaigns. Changing the code quality requires better code obfuscation kits (that do not simply produce dead code and variables) or even manual obfuscation, effectively raising the bar for the effort necessary to avoid detection.

4.1.2 String Obfuscation

Strings are heavily used by both benign and malicious applets. However, the way these strings are used differs according to their purpose. While strings in benign applications are mainly used to display text to the user, strings in malicious applets are used for obfuscation and to store machine-readable shellcode. Such strings are often chopped up and stored in the data section of the applet, and then reconstructed at run time, to be used as function arguments, class names, function names or file names. The reason for string obfuscation is to defend against signature-based systems. We have features to model the length of strings in the constant pool, their number, and the fraction of strings that contain characters that are not ASCII printable. For the length feature, we determine the length of the shortest and longest string in the pool as well as the average length of all strings.

4.1.3 Active Code Obfuscation

The features in this category characterize code constructs that are often used to thwart static code analysis. These constructs are different in that they do not attempt to hide strings (or code fragments) from manual analyst or signature-based systems, but, instead, try to mislead static analysis.

To counter code analysis techniques that check for the invocation of known vulnerable library functions within the Java library, malicious applets frequently use reflection, a part of the Java language that allows for a program to modify its structure and behavior at runtime, to invoke these functions indirectly. Other (malicious) applets use reflection to instantiate objects whose type cannot be determined statically. If an object type is not known, it is generally impossible to statically decide which method is called on it. Additionally, malicious applets sometimes dynamically load parts of the exploit at runtime or use a key, passed in as a parameter, to decrypt shellcode. To detect such activity, we count the absolute number of times reflection is used in the bytecode to instantiate objects and to call functions. In addition, we check if the `Java.io.Serializable` interface is implemented by any of the classes belonging to the applet. The reason is that some malicious applets load a serialized object at runtime that contains otherwise hidden parts of the exploit.

In addition to using reflection, some malicious applets avoid instantiating objects (instances) of specific types (subclasses) altogether. While there is some legitimate use for instantiating instances of the base class `java.lang.Object`, this is not very common in benign applets, as programmers do not want to lose the benefits of a strongly-typed language. We check if arrays or single instances of `java.lang.Object` or `java.lang.Class` are instantiated.

It is also possible to interpret JavaScript from within applets, via a Java library routine. This is highly uncommon in benign applets, but used by malicious ones to carry out general purpose computation outside the scope of most Java analysis tools. Therefore, we check if the JavaScript interface is used.

4.2 Behavioral Features

The overwhelming majority of applet malware has a specific purpose: Escaping the Java sandbox to infect the end user's machine and make it a member of a botnet. The features in this section aim at statically identifying this behavior.

4.2.1 Interaction with Runtime and Security System

As described previously, the JVM sandbox restricts applets by limiting the accessible resources. Several vulnerabilities in different versions of the Sun Java plugin have led to exploits that bypass the sandboxing mechanisms. Most of these exploits need to interact with the Java security mechanisms in different ways to be successful. Therefore, we collect features related to these security mechanisms. More precisely, we check if an applet calls into the Java runtime system (by using the `Runtime` class), interacts with the system security manager, or accesses system properties. We also check if the applet uses the `System` class to load additional code from the client system. Furthermore, we check whether the applet extends the `ClassLoader` class in an attempt to introduce its own class loader, or implements the `ClassLoaderRepository`, `SecurityPolicy`, or `PrivilegedAction` interfaces. These mechanisms are used to elevate applet privileges at runtime and can introduce code that runs without sandbox restrictions.

Features that cover these sensitive core classes of the Java security mechanisms allow for anomaly detection of new exploits. The reason is that it is likely that new exploits will need to use these mechanics in some way to break out of the sandbox.

4.2.2 Download and Execute

We characterize the download behavior of applets by checking whether `java.net.URL` objects are instantiated or sockets are used. We also check for the use of functions that are able to write files. For a successful exploit, it is necessary to execute a file after it has been downloaded. There exist a number of different ways in which the Java library API can be used to spawn a new process from a binary, e.g., by using the `java.awt.Desktop` class, or, again, the system interface. We have manually inspected all classes known to us that are able to spawn a new process and collected the API functions implementing this functionality. Since this kind of functionality is offered only by the Java library, we consulted its documentation for the relevant classes and listed the functions. We check if any of these functions are potentially used.

By detecting all known possible ways for an applet to spawn a new process, we make it impossible for malicious applets to achieve this without triggering the corresponding detection feature.

4.2.3 Jar Content

Benign applets are usually written with the goal of displaying something to the visitor of a web page or to play audio. Typically, the files necessary to do so (media files, images, ...) are included in the Jar archive. We count the number of files in the Jar that are not Java class files. Furthermore, we check all files in the archive to see if any of them contain binary machine code, i.e., if there is an executable or library. In addition, we use the total size of the Jar archive in bytes as a feature.

These features characterize benign properties rather than malicious ones. An attacker who attempts to mimic the structure of legitimate Jar archives is forced to increase the size of the exploit, which raises the distribution costs for successful campaigns.

4.2.4 Known Vulnerable Functions

Finally we also compiled a set of five well known vulnerable components of the Java library: Three individual functions, a combination of two functions, and one interface. The first two functions are `MidiSystem.getSoundbank` and the constructor for `javax.management.remote.rmi.RMIConnectionImpl`. We will discuss the third function separately, since it needs some more explanation. The combination of functions is `MidiSystem.getSequencer`, and `Sequencer.addControllerEventListener`. Additionally, the `javax.management.MBeanServer` interface can be implemented by an applet to exploit a known vulnerability.

The third, individual function that we check for is usually used in MIDlet scams. MIDlets are applications that use the Mobile Information Device Profile (MIDP) of the Connected Limited Device Configuration (CLDC) for the Java Micro Edition, typically used on cellular phones. Most of these MIDlets implement games or display a specific web page to the user. A vulnerability existed in certain phones that allowed tricking the user into sending text messages to expensive pay-per-call numbers. To this end, attackers had to call the text message send function of Java ME: `MessageConnection.send`. Note that apart from that specific vulnerability, there are a lot of MIDlets trying to trick the user into sending SMS by social engineering means. Java ME is no longer used in modern smart phones, such as the iPhone or Android. Older phones are not very valuable targets for infection, except for the mentioned text message scam.

Exploits that target well-known, vulnerable Java library functions have to call the aforementioned functions to be successful. We introduced five Boolean features, each of which is set to true if we find at least one call to one of the three functions, the combination of the two functions, or a class that implements the vulnerable interface. Of course, these functions (and interfaces) also serve a legitimate purpose. However, we found that they are rarely used by benign applets (and our other features help in making this distinction). To assert this we looked at more than 200 benign applets manually, we decompiled them and inspected their source code to check for malicious behavior and whether the description (if available) of their intended purpose matched what we found in the code. The benign applets we looked at were from a random crawl of the Internet, and two sites offering collections of applets. These data sets are described in more detail in the evaluation chapter. Their rare usage probably also indicates that they were not well-tested in the first place and thus the vulnerabilities were overlooked. If more vulnerable functions were to be added to this set later on, we would expect them again to be somewhat obscure and rarely used by benign applets. These few functions provide a way to break out of the sandbox without user interaction in vulnerable Java versions.

4.3 Feature Discussion

Jarhead collects a total of 42 features: Six are numeric (real values), ten are integers, and the remaining 26 are Boolean values. In the following paragraphs, we argue that our features are robust, in the sense that they make it difficult for an attacker to evade our analysis, while still achieving his goal. Furthermore we examine the usefulness of individual feature groups and list our top ten features.

Our features fall into two categories that capture both the behavioral and the obfuscation aspects of malicious applets. If an attacker tries to evade the obfuscation features, he will more likely be caught by traditional signature-based detection systems. An effort could be made to improve obfuscation techniques to match the code metrics of benign code very closely. However, in the presence of shellcode detection systems [12, 14], the shellcode within the applets, which is typically encoded in strings, needs to be well hidden. Furthermore, while active obfuscation techniques, such as reflection, are limiting static analysis, their use is not common in benign applications and hence, might even serve as indicators for maliciousness.

Even if an attacker finds a way to evade the obfuscation features, in order to achieve his purpose of infecting end users, his malware will necessarily implement functionality to achieve this goal. That is, the attacker needs to download and start malware on the victim's PC, and to do this, the code needs to break out of the Java sandbox. By capturing this behavior, intrinsic to the attacker's goal, we make it difficult for malicious code to evade our analysis and still succeed.

Previous systems [11, 13] for the detection of web-based exploits also use features to detect obfuscation and malicious behaviors. However, the concrete instances of these features are quite different. The main reason for these differences is that Jarhead uses static code analysis. Previous work on the detection of malicious JavaScript and Flash, on the other hand, relies on dynamic analysis. As a result, our novel features compute the presence and frequency of certain functions and resources, code patterns, and artifacts over the entire code base. Dynamic systems, on the other hand, typically compute the number of times certain events or activities occur during the execution of a program.

Answering the question how much impact do our top ten features (Table 1) have, compared to the full feature set we used a classifier with ten fold cross validation. We ran the classifier once with all our features and then with only the top ten features enabled on a total of 3372 applets, combining all our data sets. Again, the data sets are described in the evaluation chapter in detail along with more experiments. This experiment was run after the experiments in the evaluation section due to demand by our reviewers. Out of the 3372 applets the classifier working only with our top ten features misclassified a total of 122 (3.6%) applets, while our full feature set misclassified only 35 (1.0%) applets. This shows, that while our top ten features already perform reasonably well, our other features help to achieve even better results.

Merit	Attribute	Type
0.398	gets_parameters	behavior
0.266	functions_per_class	obfuscation
0.271	no_of_instructions	obfuscation
0.257	gets_runtime	behavior
0.254	lines_of_disassembly	obfuscation
0.232	uses_file_outputstream	behavior
0.22	percent_unused_methods	obfuscation
0.211	longest_string_char_cnt	obfuscation
0.202	mccabe_complexity_avg	obfuscation
0.197	calls_execute_function	behavior

Table 1: The ten most useful features, sorted by average merit.

In another experiment, we wanted to see how well the group of the obfuscation features performs without the behavioral features enabled and vice versa. To this end we used the same 3372 applets as before with ten fold cross validation, once with only the behavioral features and once with only the obfuscation features enabled. While the obfuscation features and behavioral features both performed reasonably well by themselves, with 119 (3.5%) respectively 150 (4.5%) misclassified applets they work best when used together. As stated before with all features enabled, we misclassify only 35 (1.0%) of the applets.

5. EVALUATION

In this section, we evaluate our system to measure how well it performs in detecting malicious applets. For our experiments, we used two different datasets: A manually-compiled dataset of applets (the manual set), and a set of samples collected and provided to us by the authors of the Wepawet system (the wepawet set).

5.1 Results: Manual Dataset

The manual dataset contains samples from four different sources: Two online collection of applets, an archive of mostly-malicious applets, and a number of manually collected samples. The two online collections[1] offer a selection of Java applets for web designers to use in their web pages. We crawled these two pages and downloaded a total of 1,002 Jar files. In addition, we obtained an archive of 357 Jar files with unknown toxicity from a malware research community site[2]. Finally, we manually searched for malicious applet samples on security sites[3], and we wrote a crawler that we seeded with Google search results on a combination of the terms "Java," "applet," and "malicious." This yielded another 1,495 Jar files, for a total of 2,854 samples. By combining applets from a broad range of different sources in this data set we tried to ensure it is diverse, representative of applets used in the wild and not biased to a specific kind of applet.

We first cleaned the manual dataset by removing duplicates, broken Jar files, and non-applets; i.e., we removed all Jar files that, after extraction, did not contain at least one class file that was derived from one of the applet base classes (`java.applet.Applet`, `javax.swing.Japplet` or `javax.microedition.midlet.MIDlet`). After this initial filter step, there were 2,095 files (of unknown toxicity) left.

To obtain ground truth for building and evaluating our classifier, we submitted the entire manual dataset to Virustotal. Virustotal is a website that provides free checking of suspicious files using 42 different anti-virus products. Virustotal found 1,721 (82.1%) of the files to be benign and 374 (17.9%) to be malicious (we counted a sample as benign if none of the Virustotal scanners found it to be malicious and as malicious if at least one scanner found it to be malicious).

Using the results from Virustotal as a starting point, we built an initial classifier and applied it to the manual dataset. In other words, we performed training on the manual data set using the results from Virustotal as ground truth. We then manually inspected all samples for which the results from our classifier and the results from Virustotal were different. We decompiled the samples and inspected their source code checking for known exploits that use certain functions, download and execute behavior. In the cases were documentation was available for a (probably) benign applet, we checked if the described behavior was found in the source code. Often checking the checksum of a jar file on Google provided a good indication if this was a known piece of malware. If after a while of manually inspecting the applet we were still unsure, we stuck with the verdict assigned by Virustotal. In this process, we found that Virustotal has actually misclassified 61 (2.9%) applets. In particular, Virustotal classified 34 (1.6%) benign applets as malicious, and deemed 27 (1.3%) malicious applets as benign. We manually corrected the classification of these 61 applets, and used the resulting, corrected classification as the ground truth for our dataset (with 381 malicious and 1,714 benign samples).

In the next step, we trained different classifiers on the manual dataset. More precisely, the manual dataset was split into a training set and a test set, using ten-fold cross validation. We evaluated C4.5 decision trees, support vector machines, and Bayes classification, which all showed comparable results. Eventually, decision trees turned out to be the most reliable classifiers, and they also provide good explanatory capabilities.

The detection results for the decision tree classifier were very good. With ten-fold cross validation, the classifier only misclassified a total of 11 (0.5%) samples. The false positive rate was 0.2% (4 applets), the false negative rate 0.3% (7 applets).

If we compare our detection results to the results produced by Virustotal, we find that our results are both better in terms of false positives and false negatives, and we see a reduction in the total number of misclassifications by a factor of six. This is in spite of the fact that we used Virustotal to build our initial labels, before manually adjustments (and hence, our ground truth might be biased in favor of Virustotal).

An overview of the results for the manual dataset are given in Table 2.

Discussion.

We examined the 11 instances (4 false positives, 7 false negatives) in more detail: One false positive was a potentially benign MIDlet, which had the ability to send text messages to arbitrary numbers. Two false positives were triggered by applets that did show suspicious behavior by trying

[1] `http://echoecho.com` and `http://javaboutique.internet.com`
[2] `http://filex.jeek.org`
[3] `http://www.malwaredomainlist.com`

	Virustotal (42 AVs)	Jarhead (10x cross-val.)
False pos.	1.6%	0.2%
False neg.	1.3%	0.3%

Table 2: Comparison of Jarhead and Virustotal misclassifications - note that our ground truth is biased towards Virustotal.

to write local files and executing commands on the windows command shell, but they were probably not intended to be malicious. The last false positive is an obfuscated applet that is performing calculations based on the current date, without ever displaying the results; its purpose remains unclear to us.

Two false negatives were applets that contained incomplete (broken) exploits. While these applets would not successfully execute exploits, we still count them as false negatives, since their intent is malicious (and we classified them as benign). We missed three additional malicious applets that were using reflection or loading additional code at runtime to protect against static analysis and that used an instance of CVE-2010-0094 [5]. CVE-2010-0094 is a flaw in the object deserialization that allows attackers to call privileged Java functions without proper sandboxing. The last two misclassified applets were instances of CVE-2009-3869 [4]. This particular stack-based buffer overflow is hard to catch for our system, since the vulnerable function it exploits within the native JVM implementation is not directly triggered by a specific function called from the bytecode, but, instead, is reachable from a large set of widely-used Java library functions.

Despite a few misclassified instances, we found that our system performs detection with high accuracy. In particular, some of the incorrect cases are arguably in a grey area (such as possibly benign applets that try to execute commands directly on the Windows command line and malicious applets that contain incomplete, half-working exploits).

We also examined the decision tree produced by our classifier in more detail. We found that our most important features, i.e., the ones that are highest up in the decision tree, include both features from our obfuscation set and our behavioral set. The top ten features include features capturing the interaction with the runtime, the execute feature, and the feature monitoring local file access. We also find the text message send function to be important. On the obfuscation side, we find size features, string features, and the number of functions per class to be the most effective at predicting malicious behavior.

We also tried to understand the impact of removing the feature that checks for known vulnerable functions (since this could be called signature-based detection). If this feature is removed, the misclassification rate increases only marginally, by 0.17%. This confirms that many features work together to perform detection, and the system does not rely on the knowledge of specific, vulnerable functions. To further emphasize this point, we draw the attention to the ability of Jarhead to detect zero-day attacks. After we finished the first set of experiments on the manual dataset, a new vulnerability[7], which was not part of our training set, began to see widespread use. When we tested our classifier on these completely new exploits, it was able to identify all five samples we acquired that implement this exploit, without any adjustment.

5.2 Results: Wepawet Dataset

To further test our classifier on real-world data, we collected Jar files from the Wepawet system. Wepawet is a public portal that allows users to upload suspicious URLs. Wepawet would then fetch the content at these URLs and attempt to detect drive-by download exploits. Since certain drive-by download exploit kits make use of malicious Java code, we expected to find some applets that Wepawet users on the Internet might have submitted to the system. Unfortunately, Wepawet does currently not support the analysis of Jar files, and hence, would consider all such files as benign.

The authors of Wepawet provided us with 1,551 Jar files that the system had collected over the past months. We removed duplicates, broken archives, and non-applets (as before) and ended up with a set of 1,275 applets. Again, we used Virustotal to obtain some form of initial classification. Virustotal found 413 (32.4%) applets to be benign and 862 (67.6%) applets to be malicious. We then ran our classifier on this Wepawet set. Compared to Virustotal, we assigned the same verdict to 1,189 (93.3%) samples, while 86 (6.7%) samples were classified differently. More precisely, we classified 59 (4.6%) malicious applets (according to Virustotal) as benign and 27 (2.1%) benign applets (according to Virustotal) as malicious.

Manual examination of the instances with different classifications revealed interesting results, both for the false positives and the false negatives. For the false positives, we found that 19 of the 27 were clearly errors in the initial classification by Virustotal (that is, these 19 applets were actually malicious but falsely labeled as benign by Virustotal). That is, Jarhead was correct in labeling these applets as bad. Interestingly, one of these 19 samples was a malicious applet that was stealing CPU cycles from the user, visiting a web page to mine bitcoins [29]. While we had not seen such behavior before, Jarhead correctly classifies this applet as malicious.

The remaining eight samples were properly classified by Virustotal as benign, and hence, false positives for Jarhead. Seven of our eight false positives had the potential to download files to the user's disk, and five of these seven would even execute these files after downloading! Except for intended use of these applets as software installers for potentially benign programs, their behavior (and code patterns) are essentially identical to malware. The last false positive was a (likely benign) MIDlet that can send text messages to arbitrary numbers.

We then inspected the 59 applets that Virustotal labeled as malicious (while Jarhead labeled them as benign). Four programs were partial exploits that did not implement actual malicious behavior. The remaining nine were false positives by Virustotal. We do not consider these partial exploits (which are essentially incompletely packed archives) and the nine benign programs to be properly labeled by Virustotal.

The remaining 46 samples were actual malware. They were largely made up of two families of exploits for which we had no samples in our (manual) training set, which we used to build the classifier. 28 samples belonged to a family of MIDlet scam applets, and another 15 belonged to a new, heavily obfuscated version of a new vulnerability (CVE-2011-3544) that was not present in our training set. We

also missed one instance of CVE-2009-3869 [4] (discussed previously) and one other malicious MIDlet. Moreover, one exploit used a new method to introduce code that was not in our training set.

Of the 46 false negatives, we missed 44 samples (or 96%) because of limitations with the manual dataset used to train our classifier. While we were collecting features useful for identifying these malicious applets, our classifier did not learn that they were important, because it was missing samples triggering these features in its training set. To show that we can achieve better results with a better training set, and to demonstrate that our features are indeed well-selected and robust, we trained and tested a new classifier on the Wepawet dataset using ten-fold cross validation. For that experiment, we found a total misclassification count of 21 (1.6%), the total false positive rate was 0.9% (12 applets), and the false negative rate 0.7% (9 applets).

The results for the Wepawet dataset are presented in Table 3.

	Original classifier	10x cross validated
False positives	2.1%	0.9%
False negatives	4.6%	0.7%

Table 3: Jarhead's performance on the Wepawet dataset.

We also collected performance numbers during our experiments. On average our analysis takes 2.8 seconds per sample, with a median of 0.6 seconds. This shows that the majority of samples is very fast to analyze, although there is a narrow longtail for which the analysis takes longer (specifically, 2% of the samples take longer than 10 seconds and 0.3% took longer than a minute). For the slower 50% of the samples, more than 98% of the total running time was spent in the disassembler. Thus, these numbers could be significantly improved simply by a more efficient implementation of the disassembler or by implementing our feature collection directly on the bytecode so disassembling the code becomes unnecessary.

6. POSSIBLE EVASION

We have seen that Jarhead performs well on real-world data. In this section, we will discuss the limitations of Jarhead, i.e., possible ways for malicious applets to avoid detection by our system.

A lot of the usual limitations for static analysis [28, 25, 26] do not apply to Java bytecode. However, for a trusted applet, it is possible to use the Java native interface (JNI) to execute native code on the machine. This is not covered by our analysis. If the parts of the malware that are implementing the malicious behavior are in the Java bytecode parts of the applet, it is likely that we will detect them, otherwise, there exist many analysis tools for native malware that would be able to detect such malicious behavior.

Static analysis is also limited by the use of reflection in a language. Interestingly, we found that reflection is not in widespread use by benign Java applets. Malicious applets, however, use it in an attempt to evade systems such as ours. While we do not completely mitigate this problem, we have features that aim to precisely detect this kind of evasion. Moreover, other features that target the Jar file and its code as a whole, such as code metrics and Jar content features, are unaffected by reflection.

We examine each applet individually. Applets on the same web page can communicate with each other, by calling each others public methods. Applets can also be controlled from the surrounding JavaScript in a similar fashion. If malicious behavior is distributed among multiple applets within a single page, or partly carried out by the surrounding JavaScript, our analysis scope is too limited, and we might misclassify these applets. Fortunately, to the best of our knowledge, malicious applets that use JNI and malicious code splitting behavior between multiple applets (or interacting with the surrounding JavaScript) do currently not exist in the wild. Moreover, we can extend our analysis to consider multiple applets that appear on the same page together. We already combine all class files within a single Jar archive, so such an extension would be rather straightforward.

While we have shown that today's malicious applets are very well covered by our features, a completely new class of exploits or vulnerabilities could bypass our detection either because we do not collect good features to capture the new exploit or because the classifier was unable to learn this exploit pattern from the training set. In these cases, it might be necessary to add new features or extend the set of known vulnerable functions. This would be straightforward to do. In other cases, simply retraining the classifier on a dataset containing the new exploits might suffice.

Since we operate on the Java bytecode, identifying vulnerabilities in the underlying native implementation of the JVM itself (such as CVE-2009-3869 [4]) is difficult. The reason is that corresponding exploits target a heap overflow vulnerability by displaying a specially crafted image. The set of possible functions within the Java API that can lead to execution of the vulnerable function is very large, and the API functions are widely used. Moreover, there is not obvious malicious activity present in the Java class file when this vulnerability is triggered.

7. RELATED WORK

A lot of research has been done to detect malware. In this section, we present different approaches and compare them to Jarhead.

Signature-based approaches [33, 2] find malware by matching it against previously selected code or data snippets specific to a certain exploit. Signature-based detection systems can be evaded by obfuscation, and cannot catch exploits they do not have signatures for. Jarhead complements signature-based techniques by identifying malicious samples based specifically on their obfuscation and behavior features. Jarhead is also able to detect previously-unknown families of exploits (since it uses anomaly detection).

A broad range of low and high interactive honeyclients were proposed to identify malware [30, 10, 11]. They cannot detect malware that targets vulnerable components that are not installed on the honeyclient. Specifically for Java applets, this means that the honeyclients need to have the correct version of the Java plugin installed, running in the correct configuration with the correct browser. Jarhead is able to detect malicious applets independent of this environment by using static analysis. Furthermore, the runtime environment of honeyclients can be fingerprinted by malware as part of evasion attempts. Since Jarhead relies purely on

static analysis, fingerprinting the analysis system is not possible.

Twelve years ago, Helmer suggested an intrusion detection system aimed at identifying hostile Java applets [18]. Their system is geared towards the detection of applets annoying the user, rather than real malicious ones as we see today. Their approach is also based on machine learning combined with anomaly detection, but the features are very different. In particular, their system monitors the system call patterns emitted from the Java runtime system during applet execution. The system has not been tested on real malicious applets and requires dynamic execution, exposing it to similar problems as honeyclients. Jarhead has been tested on a large real-world dataset of modern, malicious applets, and it is not subject to the limitations that come with dynamic malware execution.

8. CONCLUSIONS

We address the quickly growing problem of malicious Java applets by building a detection system based on static analysis and machine learning. We implemented our approach in a tool called Jarhead and tested it on real-world data. We also deployed our system as a plugin for the Wepawet system, which is publicly accessible. Our tool is robust to evasion, and the evaluation has demonstrated that it operates with high accuracy.

In the future, we plan to improve our results by using more sophisticated static analysis techniques to achieve even higher accuracy. For example, we would like to use program slicing [20] to statically determine whether a downloaded file is indeed the one that is executed later in the program or whether a suspicious image file is actually passed to a vulnerable function.

9. ACKNOWLEDGMENTS

This work was supported by the Office of Naval Research (ONR) under Grant N000140911042, by the National Science Foundation (NSF) under grants CNS-0845559 and CNS-0905537, and by Secure Business Austria.

10. REFERENCES

[1] Capture hpc. http://nz-honeynet.org.
[2] Clamav. http://www.clamav.net.
[3] CVE-2009-3867. National Vulnerability Database.
[4] CVE-2009-3869. National Vulnerability Database.
[5] CVE-2010-0094. National Vulnerability Database.
[6] CVE-2010-0842. National Vulnerability Database.
[7] CVE-2012-0507. National Vulnerability Database.
[8] Wepawet. http://wepawet.iseclab.org.
[9] Same origin policy. http://www.w3.org/Security/wiki/Same_Origin_Policy, 2010.
[10] Yaser Alosefer and Omer Rana. Honeyware: A web-based low interaction client honeypot. ICSTW '10, 2010.
[11] Marco Cova, Christopher Kruegel, and Giovanni Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *World-wide web conference (WWW)*, 2010.
[12] Manuel Egele, Peter Wurzinger, Christopher Kruegel, and Engin Kirda. Defending browsers against drive-by downloads : mitigating heap-spraying code injection attacks. In *DIMVA'09*, 2009.
[13] Sean Ford, Marco Cova, Christopher Kruegel, and Giovanni Vigna. Analyzing and Detecting Malicious Flash Advertisements. In *Annual Computer Security Applications Conference (ACSAC)*, 2009.
[14] Y. Fratantonio, C. Kruegel, and G. Vigna. Shellzer: a tool for the dynamic analysis of malicious shellcode. In *Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID)*.
[15] Mike Geide. 300% increase in malicious jars. http://research.zscaler.com/2010/05/300-increase-in-malicious-jars.html, 2010.
[16] Li Gong and Marianne Mueller e.a. Going beyond the sandbox: An overview of the new security architecture in the java development kit 1.2. USITS, 1997.
[17] Hall and Mark e.a. The weka data mining software: an update. *SIGKDD Explor. Newsl.*, 11(1), 2009.
[18] Guy G. Helmer and Johnny S. Wong e.a. Anomalous intrusion detection system for hostile java applets. *Journal of Systems and Software*, 55(3), 2001.
[19] Stefanie Hoffman. Microsoft warns of unprecedented rise in java exploits. http://www.crn.com/news/security/227900317/microsoft-warns-of-unprecedented-rise-in-java-exploits.htm, 2010.
[20] Susan Horwitz and Thomas Reps e.a. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12, 1990.
[21] Wolfgang Kandek. The inconvenient truth about the state of browser security. http://laws.qualys.com/SPO1-204_Kandek.pdf, 2011.
[22] Brian Krebs. Java: A gift to exploit pack makers. http://krebsonsecurity.com/2010/10/java-a-gift-to-exploit-pack-makers, 2010.
[23] Brian Krebs. Exploit packs run on java juice. http://krebsonsecurity.com/2011/01/exploit-packs-run-on-java-juice/, 2011.
[24] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.
[25] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security*, CCS '03, 2003.
[26] Douglas Low. Java control flow obfuscation. Technical report, 1998.
[27] Thomas J. McCabe. A complexity measure. *IEEE Trans. Software Eng.*, 2(4), 1976.
[28] Andreas Moser and Christopher Kruegel e.a. Limits of static analysis for malware detection. In *ACSAC*, 2007.
[29] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. 2008.
[30] Jose Nazario. Phoneyc: a virtual client honeypot. LEET'09, 2009.
[31] Niels Provos and McNamee e.a. The ghost in the browser analysis of web-based malware. In *Proceedings*

of the first conference on First Workshop on Hot Topics in Understanding Botnets, HotBots'07, 2007.

[32] Niels Provos and Panayiotis Mavrommatis e.a. All your iframes point to us. *Google Inc*, 2008.

[33] Martin Roesch. Snort - lightweight intrusion detection for networks. LISA '99, 1999.

[34] Christian Seifert and Ian Welch e. a. Identification of malicious web pages through analysis of underlying dns and web server relationships. In *LCN*, 2008.

Hi-Fi: Collecting High-Fidelity Whole-System Provenance

Devin J. Pohly, Stephen McLaughlin, Patrick McDaniel
Pennsylvania State University
University Park, PA
{djpohly,smclaugh,mcdaniel}@cse.psu.edu

Kevin Butler
University of Oregon
Eugene, OR
butler@cs.uoregon.edu

ABSTRACT

Data provenance—a record of the origin and evolution of data in a system—is a useful tool for forensic analysis. However, existing provenance collection mechanisms fail to achieve sufficient breadth or fidelity to provide a holistic view of a system's operation over time. We present Hi-Fi, a kernel-level provenance system which leverages the Linux Security Modules framework to collect high-fidelity whole-system provenance. We demonstrate that Hi-Fi is able to record a variety of malicious behavior within a compromised system. In addition, our benchmarks show the collection overhead from Hi-Fi to be less than 1% for most system calls and 3% in a representative workload, while simultaneously generating a system measurement that fully reflects system evolution. In this way, we show that we can collect broad, high-fidelity provenance data which is capable of supporting detailed forensic analysis.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—*Security and protection*; D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software*

General Terms

Security, Design

Keywords

data provenance, forensics, malware, reference monitor

1. INTRODUCTION

Data provenance, which is a detailed record of the origin and evolution of data in a system, is a useful tool in systems security. In its raw form, provenance data is simply a series of system events, such as a file being written or a process being created. Taken together, these events form the provenance record for that system, and examining this record can reveal detailed information about the system's secure or insecure operation. Previous works on data provenance have pointed out many such possibilities, such as performing intrusion detection [19] or identifying data which may have been exfiltrated from the system [10].

Provenance records are well suited to system forensics. Current forensic analysis techniques exploit the flexibility of event-based logs for a number of purposes. For example, audit logs can be used to evaluate ongoing compliance with real-world policies [6] or to create detailed reconstructions of several aspects of system state [7]. A complete provenance record provides an even richer set of information for this purpose (see Section 3.2).

However, for a data provenance system to provide the holistic view of system operation required for such forensic applications, it must be complete and faithful to actual events. This property, which we call "fidelity," is necessary for drawing valid conclusions about system security. A missing entry in the provenance record could sever an important information flow, while a spurious entry could falsely implicate an innocuous process. As we discuss in Section 2.1, these requirements can be achieved by designing the provenance collection mechanism around the reference monitor concept [1]. In particular, this mechanism must provide complete mediation for events which should appear in the record.

The following scenario illustrates this need: Alice runs a high-profile website. One day, her web server is infected by the (hypothetical) PwnHP worm. PwnHP takes control of a website's behavior by infecting the system's PHP binary. It also starts a daemonized process which periodically connects to a command and control server for instructions. These connections alert Alice to the fact that something is amiss.

Fortunately, Alice is collecting provenance data for this system. She retrieves the logs from her append-only storage server and begins to investigate. First, she locates one of the outgoing connections in the provenance record and traces the process provenance back to the original compromised thread in her web server. She can then follow the provenance trail forward to see the modified PHP binary, as well as all of the malicious behavior that it performs when executed. Alice can then proceed with confidence in restoring her system to a good state.

One lesson we can learn from this story is that forensic investigation requires a definition of provenance which is broader than just file metadata. What is needed is a record of *whole-system provenance* which retains actions of processes, IPC mechanisms, and even the kernel. These "transient" system objects can be meaningful even without being an ancestor of any "persistent" object. The command-and-control daemon on Alice's server, for example, was significant because it was a *descendant* of the compromised process. If the provenance system had deemed it unworthy of inclusion in the record, she could not have traced the outgoing connections to the compromise.

ACSAC '12 Dec. 3–7, Orlando, Florida USA

In this paper, we present Hi-Fi, a provenance system designed to collect high-fidelity whole-system provenance. Hi-Fi is the first provenance system which can collect a *complete* provenance record from early kernel initialization through system shutdown. Unlike existing provenance systems, it accounts for all kernel actions as well as application actions. Hi-Fi can also collect *socket provenance*, creating a system-level provenance record that spans multiple hosts. Furthermore, it solves a number of design and implementation problems unique to this work.

We evaluate Hi-Fi in two ways. First, we demonstrate its ability to capture behavior on a system running malicious software. We create a tool which performs common malicious actions such as creating a backdoor account, establishing persistence in the system, and exfiltrating sensitive data. In each case, inspection of the system provenance record revealed the malicious actions. Second, we evaluate Hi-Fi's performance for individual system calls and for a system-call heavy workload. We observe an overhead of less than 1% for most system calls, and a maximum of 6% for the `read` system call. For an I/O-bound workload, the average overhead is less than 3%.

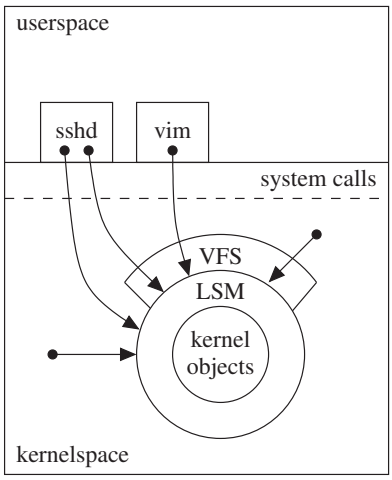


Figure 1: Complete mediation with LSM

2. BACKGROUND

Maintaining provenance records is a well-established practice in fields which deal with physical artifacts, but provenance for digital artifacts is a comparatively new application. The earliest implementations of digital provenance focused on highly structured, special-purpose data. One such system is Trio [25], a database management system that stores the provenance of its records. Many other special-purpose systems exist, such as Panda [9], which focuses on specific workflows, and provenance aware Condor [18], which collects provenance for jobs on a specific batch system.

To support forensic analysis, however, we need the ability to trace arbitrary, unstructured data. This requires general-purpose, system-level provenance collection. The first such provenance system, Lineage File System [19], accomplished this by intercepting system calls in a modified Linux kernel. When an application executed one of these calls, a record describing the action would be written to the `printk` buffer and stored in a MySQL database. The same system-call approach is used by more recent systems, such as PASSv2 [15], which handles ten different system calls, and Forensix [8], which intercepts around seventy-five. These systems analyze the arguments to system calls and write provenance data to log files on the disk. Unfortunately, system-call interception cannot produce a complete provenance record, because the kernel itself does not use system calls. Kernel-initiated actions, such as executing the interpreter for Alice's PHP scripts, are therefore not captured at the system-call layer.

Another option for collecting system-level provenance is to instrument the filesystem layer (e.g., the Linux VFS). This is the approach taken by the Story Book provenance system [21]. Story Book is designed as a framework which allows multiple "provenance sources" to collect data. One of the provided sources is a filesystem implemented using the FUSE API [4]. This filesystem acts as a layer between the kernel and an existing filesystem, capturing file activity and writing the resultant provenance data to a custom transactional storage system. Both the kernel and applications access files through the VFS, so this approach can generate a complete record of file activity. However, this does not provide the *whole-system* view of provenance needed for forensics.

2.1 Linux Security Modules

In order to create a system which does provide the needed properties, we draw on the three design goals of the reference monitor concept [1]. Tamperproofness, which states that a system cannot be made to behave incorrectly, will ensure that our provenance collector does not generate spurious or inaccurate records. Complete mediation guarantees that every access is handled, whether initiated by an application or by the kernel. For a provenance collector, this ensures that every legitimate event will appear in the record. Finally, if the collector is simple enough to be verified, then we can be certain that the first two properties hold. Taken together, these three conditions guarantee fidelity of the provenance record.

Current approaches to provenance systems do not provide sufficient fidelity. Our system overcomes this by building on a framework intended for complete mediation. Linux Security Modules, or LSM, is a framework which was originally designed for integrating custom access control mechanisms into the Linux kernel [26]. It does this by mediating access, not to system calls, but to kernel objects themselves, as Figure 1 illustrates. The LSM framework comprises a set of hooks which are carefully placed throughout the kernel. Security modules can provide an implementation for any of these hooks, which are executed just before the corresponding access takes place. The placement of these hooks has been repeatedly analyzed and refined [3, 5, 28, 24] to ensure that every access is mediated.

The designers of the LSM framework are deliberate in establishing where this mediation takes place. In particular, they identify several issues with system-call interception: that it "is not race-free, may require code duplication, and may not adequately express the full context needed to make security policy decisions" [26]. LSM was created to avoid these problems and provide complete mediation, which is required for high-fidelity provenance collection.

3. DESIGN

Hi-Fi consists of three components: the provenance collector, the provenance log, and the provenance handler. Figure 2 depicts the interaction between these components. The collector is an LSM; as such, it resides in kernelspace and is notified whenever a kernel object access is about to take place. When invoked, the collector constructs an entry describing the action and writes it to the provenance log. The log is a buffer which presents these entries to userspace as a file. The provenance handler can then access this file using the standard file API, process it, and store the provenance record. The handler used in our experiments simply copies the log data to a file on disk, but it is possible to implement a custom handler for any

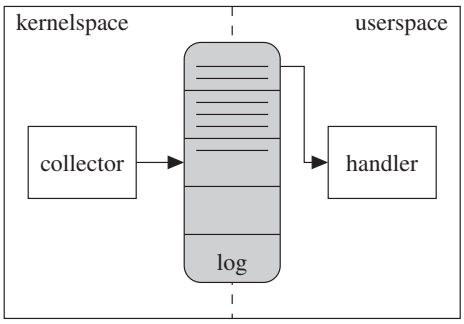


Figure 2: Architecture of Hi-Fi

purpose, such as post-processing, graphical analysis, or storage on a remote host.

3.1 Threat Model

We define a threat against our system as any way of compromising the fidelity of the provenance record during collection. Hi-Fi maintains the fidelity of provenance collection under any userspace compromise. This is a strictly stronger guarantee than those provided by current system-level provenance collection systems. In the event of a kernel-level compromise, the adversary will be able to tamper with the compenents of the provenance collector. However, the integrity of data *up to and including* the kernel compromise can be protected by an isolated disk-level versioning system [22] or a strong write-once read-many storage system [20]. In fact, since provenance data never changes after being written, a storage system with strong WORM guarantees is particularly well-suited to this task. For socket provenance, Hi-Fi guarantees that incoming data will be recorded accurately; to prevent on-the-wire tampering by an adversary, standard end-to-end protection such as IPsec should be used.

3.2 Provenance Collector

The main component of Hi-Fi is the in-kernel provenance collector, which is responsible for observing provenance-generating events. The collector consists of a number of LSM hooks which mediate operations on kernel objects. Table 1 lists all of the hooks which generate provenance data; several other hooks are used for internal memory management. For each hook, the collector gathers the relevant context from the kernel and writes one or more entries to the provenance log. By mediating the appropriate kernel objects, we are able to capture a wide variety of events:

- Reads and writes to file descriptors, including regular files, device files, and pipes.
- File operations: renaming, changing permissions, etc.
- Inter-process communication, such as shared memory, message queues, and UNIX domain sockets.
- Network communication between provenanced hosts.
- Program execution with full arguments and environment.
- Creation and deletion of credential objects (creds), which represent both process and kernel actions.
- User transitions, e.g., `login` changing to the authenticated user and group, `passwd` escalating to root by setuid execution, or `sshd` dropping privileges.

Kernel object	LSM hook
Inode	`inode_init_security`
	`inode_free_security`
	`inode_link`
	`inode_unlink`
	`inode_rename`
	`inode_setattr`
	`inode_readlink`
	`inode_permission`
Open file	`file_mmap`
	`file_permission`
Program	`bprm_check_security`
	`bprm_committing_creds`
Credential	`cred_prepare`
	`cred_free`
	`cred_transfer`
	`task_fix_setuid`
Socket	`socket_sendmsg`
	`socket_post_recvmsg`
	`socket_sock_rcv_skb`
	`socket_dgram_append`
	`socket_dgram_post_recv`
	`unix_may_send`
Message queue	`msg_queue_msgsnd`
	`msg_queue_msgrcv`
Shared memory	`shm_shmat`

Table 1: LSM hooks used to collect provenance

These events provide a comprehensive view of a system's history, including the entire process execution tree, the complete filesystem structure, and explicit information flows that may include network communication. These features can also be reconstructed for any given point in the past.

3.3 Provenance Handler

The responsibility of the provenance handler is to interpret, process, and store the provenance data after it is collected, and it should be flexible enough to support different needs. Consider the following examples. Alice, the website administrator we met earlier, has a dedicated provenance storage server with a huge disk. She does not want to do any extra processing or storage on her already overloaded web server; she just wants to move the provenance data over the LAN to her storage server as quickly as possible. Bob, on the other hand, is a provenance-curious researcher who would like to gather data from a number of volunteers. He would like the data formatted according to the Open Provenance Model [14] and uploaded to his web server in XML format. Alice and Bob have very different processing and storage needs for their provenance data. With an existing provenance system, their data would be stored in a database on disk before they could choose how to handle it.

Hi-Fi does not impose such limitations. Instead, we decouple the provenance handler from the collection process, allowing the system administrator to implement the handler according to the needs of the system. In our example, Alice can create a simple Bash script which pipes provenance data through `ssh` directly to her storage server. Bob is free to create a more complex handler which reads the log, uses a Java library from the OPM website to build the model and convert it to XML, and executes an HTTPS request to submit the document to his online database. He can then distribute this program to his volunteers.

An added benefit of this design is that it keeps complex algorithms out of the collector. Existing systems have devoted considerable effort to dealing with problems in provenance representation, such as compact storage or graph cycles [16]. Our design simply allows the handler to address these problems in whatever way is most appropriate.

4. SYSTEM-LEVEL OBJECT MODEL

Collecting system-level provenance requires a clear model of system-level objects. For each object, we must first describe how data flows into, out of, and through it. Next, we identify the LSM hooks (listed in Table 1) which mediate data-manipulating operations on that object, or we place new hooks if the existing ones are insufficient. Finally, we decide how the relevant objects can be uniquely identified in the provenance log.

Each entry in the provenance log describes a single action on a kernel object. This includes the type of action, the subject, the object, and any appropriate context. For example, starting a kernel build could generate the following entry:

Type	Execute
Subject	Credential 508
Object	Root filesystem, inode 982
Arguments	"make", "–j8", "bzImage"
Environment	"HOME=/home/alice", "PATH=/usr/bin:/bin", "SHELL=/bin/bash", ...

For the purposes of recording provenance, each object which can appear in the log must be assigned an identifier which is unique for the lifetime of that object. Some objects, such as inodes, are already assigned a suitable identifier by the kernel. Others, such as sockets, require special treatment. For the rest, we generate a "provid," a small integer which is reserved for the object until it is destroyed. These provids are managed in the same way as process identifiers to ensure that two objects cannot simultaneously have the same provid. When an object which needs an identifier is created, we allocate a provid and attach it using the opaque `security` pointer provided by LSM. When the object is freed, we release the provid to be used again.

In later sections, we will show log entries in an abbreviated, human-readable form, with inode numbers resolved to filenames, and forks implied by a change in the bracketed provid:

```
[508] exec rootfs:/usr/bin/make -j8 bzImage
```

4.1 System, Processes, and Threads

Our model of data flow includes transferring data between multiple systems or multiple boots of a system. We therefore need to identify each boot separately. To ensure that these identifiers do not collide, we create a random UUID at boot time. We then write it to the provenance log so that subsequent events can be associated with the system on which they occur.

Within a Linux system, the only actors are processes[1] and the kernel. These actors store and manipulate data in their respective address spaces, and we treat them as black boxes for the purpose of provenance collection. Most data flows between processes use one of the objects described in subsequent sections. However, several actions are specific to processes: forking, program execution, and changing subjective credentials.

[1]On Linux, threads are a special case of processes, so we will use the term "process" to refer collectively to both.

Since LSM is designed to include kernel actions, it does not represent actors using a PID or `task_struct` structure. Instead, LSM hooks receive a `cred` structure, which holds the user and group credentials associated with a process or kernel action. Whenever a process is forked or new credentials are applied, a new credential structure is created, allowing us to use these structures to represent individual system actors. As there is no identifier associated with these `cred` structures, we generate a provid to identify them.

4.2 Files and Filesystems

Regular files are the simplest and most common means of storing data and sharing it between processes. Data enters a file when a process writes to it, and a copy of this data leaves the file when a process reads from it. Both reads and writes are mediated by a single LSM hook, which identifies the the actor, the open file descriptor, and whether the action is a read or a write. Logging file operations is then straightforward.

Choosing identifiers for files, on the other hand, requires some thought. We must consider that files differ from other system objects in that they are persistent, not only across reboots of a single system, but also across systems (like a file on a portable USB drive). Because of this, it must be possible to uniquely identify a file independent of any running system. In this case, we can make use of identifiers which already exist rather than generate new ones. Each file has an inode number which is unique within its filesystem. If we combine this with a UUID that identifies the filesystem itself, we obtain a suitable identifier that will not change for the lifetime of the file. UUIDs are generated for most filesystems at creation, and we generate random UUIDs for the Linux kernel's internal pseudo-filesystems when they are initialized. We can then use the combination of UUID and inode number to identify the file in all filesystem operations, as well as to identify a program file when it is being executed.

4.3 Memory Mapping

Files can also be mapped into one or more processes' address spaces, where they are used directly through memory accesses. This differs significantly from normal reading and writing in that the kernel does not mediate accesses once the mapping is established. We can only record the mapping when it occurs, along with the requested access mode (read, write, or both). Note that this does not affect our notion of complete mediation if we conservatively assume that flows via memory-mapped files take place whenever possible.

Shared memory segments are managed and interpreted in the same way. POSIX shared memory is implemented using memory mapping, so it behaves as described above. XSI shared memory, though managed using different system calls and mediated by a different LSM hook, also behaves the same way, so our model treats them identically. In fact, since shared memory segments are implemented as files in a temporary filesystem, their identifiers can be chosen in the same way as file identifiers.

4.4 Pipes and Message Queues

The remaining objects have stream or message semantics, and they are accessed sequentially. In these objects, data is stored in a queue by the writer and retrieved by the reader. The simplest such object is the pipe, or FIFO. Pipes have stream semantics and, like files, they are accessed using the `read` and `write` system calls. This interaction is illustrated in Figure 3a. Since a pipe can have multiple writers or readers, we cannot represent it as a flow directly from one process to another. Instead, we must split the flow into two parts, modeling the data queue as an independent file-like

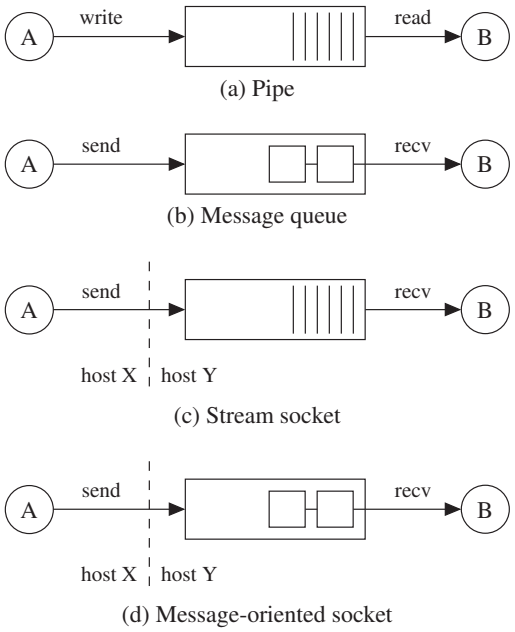


Figure 3: Models of data flow

object. In this way, a pipe behaves like a sequentially-accessed regular file. In fact, since named pipes are inodes within a regular filesystem, and unnamed pipes are inodes in the kernel's "pipefs" pseudo-filesystem, we can choose pipe identifiers exactly as we do for files.

Message queues are similar to pipes, with two major semantic differences: the data is organized into discrete messages instead of a single stream, and these messages can be delivered in a different order than that in which they are sent. Fortunately, LSM handles messages individually, so we can create a unique identifier for each. We can then reliably tell which process receives it, regardless of the order in which the messages are dequeued. Since individual messages have no natural identifier, we generate a provid for each.

4.5 Sockets

Sockets are the most complex form of inter-process communication handled by our system, but they can be modeled very simply. As with pipes, we treat a socket's receive queue as an intermediary file between the sender and receiver, as shown in Figure 3. Sending data, then, is just writing to this queue, and receiving data is reading from it. The details of network transfer are hidden by the socket abstraction, so we only need to consider the semantic differences between socket types.

Stream sockets provide the simplest semantics with respect to data flow: they behave identically to pipes. Since stream sockets are necessarily connection-mode, all of the data sent over a stream socket will arrive in the same receive queue. If we assign one identifier to each socket endpoint, we can use these identifiers for the lifetime of the socket. Message-oriented sockets, on the other hand, do not necessarily have the same guarantees. They may be connection-mode or connectionless, reliable or unreliable, ordered or unordered. We only know that any messages which are delivered are delivered intact. Each packet therefore needs a separate identifier, since we cannot be sure at what endpoint it will arrive.

In determining how identifiers are chosen, we must reason carefully about socket behavior. We should never reuse an identifier, since a datagram can have an arbitrarily long lifetime. We also want the identifier to be associated with the originating host. The per-boot UUID described in Section 4.1 addresses both of these requirements. By combining this UUID with an atomic counter, we can generate unique identifiers for socket provenance. As long as this counter is large enough to avoid rolling over, we can be reasonably certain that socket identifiers will remain unique.

In order to generate useful log entries, we must consider the sequence of events for sending and receiving data. Suppose process A on host X sends data which arrives in queue Q on host Y. Process B on host Y then receives this data. In this case, the following events should occur:

1. Process A passes data to the `send` function. X writes "A sends to Q" to the log.
2. The data is encapsulated in a packet as defined by the socket's protocol family.
3. The packet may be transmitted over a network.
4. The packet is either dropped, in which case no flow takes place, or it is delivered and saved to queue Q.
5. Process B is given some data from this queue as the output from the `recv` function. Before returning control to B, Y writes "B receives from Q" to the log.

Writing the "send" entry is the tricky step, because the sender needs to know the identifier of the remote receive queue. However, the sender and receiver may not have any shared information which can be used to agree on an identifier, so the cleanest solution is for the *sender* to choose an identifier for the remote receive queue and transmit it along with the first data packet. (How this happens depends on the socket's protocol family.) In this way, both the sender and receiver have the data needed to write their log entries.

5. IMPLEMENTATION DETAILS

In the course of creating Hi-Fi, we have overcome a variety of implementation challenges. Several of our solutions, such as running a provenance-opaque process, are new to the literature. Others, such as moving data efficiently from the kernel to userspace, are new solutions to problems that existing provenance work has solved in other ways.

5.1 Efficient Data Transfer

Provenance collection has been noted to generate a large volume of data [2]. Because of this, we need an efficient and reliable mechanism for making large quantities of kernel data available to userspace. Other systems have accomplished this by using an expanded `printk` buffer [19], writing directly to on-disk log files [15], or using FUSE [21]. However, none of these methods is appropriate for our system design. Instead, we use a Linux kernel object known as a "relay," which is designed specifically to address this problem [27].

A relay is a kernel ring buffer made up of a set of preallocated sub-buffers. Once the relay has been initialized, the collector writes provenance data to it using the `relay_write` function. This data will appear in userspace as a regular file, which can be read by the provenance handler. Since the relay is backed by a buffer, it retains provenance data even when the handler is not running, as is the case during boot, or if the handler crashes and must be restarted.

Since the number and size of the sub-buffers in the relay are specified when it is created, the relay has a fixed size. Although the collector can act accordingly if it is about to overwrite provenance which has not yet been processed by the handler, it is better

to avoid this situation altogether. To this end, we allow the relay's size parameters to be specified at boot time.

5.2 Early Boot Provenance

The Linux kernel's boot-time initialization process consists of setting up a number of subsystems in sequence. One of these subsystems is the VFS subsystem, which is responsible for managing filesystem operations and the kernel's in-memory filesystem caches. These caches are allocated as a part of VFS initialization. They are then used to cache filesystem information from disk, as well as to implement memory-backed "pseudo-filesystems" such as those used for pipes, anonymous memory mappings, temporary files, and relays.

The security subsystem, which loads and registers an LSM, is another part of this start-up sequence. This subsystem is initialized as early as possible, so that boot events are also subject to LSM mediation. In fact, the LSM is initialized *before* the VFS, which has a peculiar consequence for the relay we use to implement the provenance log. Since filesystem caches have not yet been allocated, we cannot create a relay when the LSM is initialized. Our design goal of fidelity makes this a problem unique to our system: not only are we forced to postpone relay setup, but we must also do so without losing boot provenance data.

We therefore separate relay creation from the rest of the module's initialization and register it as a callback in the kernel's generic "initcall" system. This allows it to be delayed until after the core subsystems such as VFS have been initialized. In the meantime, provenance data is stored in a small temporary buffer. Inspection of this early boot provenance reveals that a one-kilobyte buffer is sufficiently large to hold the provenance generated by the kernel during this period. Once the relay is created, we flush the contents of the temporary boot-provenance buffer to it and free the buffer. By doing this, we can collect and retain provenance data for a large portion of the kernel's initialization process.

5.3 Operating System Integration

One important aspect of Hi-Fi's design is that the provenance handler must be kept running to consume provenance data as it is written to the log. Since the relay is backed by a buffer, it can retain a certain amount of data if the handler is inactive or happens to crash. It is important, though, that the handler is restarted in this case. Fortunately, this is a feature provided by the operating system's `init` process. By editing the configuration in `/etc/inittab`, we can specify that the handler should be started automatically at boot, as well as respawned if it should ever crash.

We also want to collect and retain provenance data for as much of the operating system's shutdown process as possible. At shutdown time, the `init` process takes control of the system and executes a series of actions from a shutdown script. This script asks processes to terminate, forcefully terminates those which do not exit gracefully, unmounts filesystems, and eventually powers the system off. Since the provenance handler is a regular userspace process, it is subject to this shutdown procedure as well. However, there is no particular order in which processes are terminated during the shutdown sequence, so it is possible that another process may outlive the handler and perform actions which generate provenance data. Our goal of fidelity requires that we collect this provenance.

Our solution is to handle the shutdown process in the same way we would handle a crash: restart the provenance handler. We modify the shutdown script to re-execute the handler after all other processes have been terminated, just before filesystems are unmounted. For this special case, we implement a "one-shot" mode in the handler which, instead of forking to the background, exits after handling the data currently in the log. This allows it to handle any remaining shutdown provenance, then return control to `init` to complete the shutdown process.

5.4 Bootstrapping Filesystem Provenance

Intuitively, a complete provenance record contains enough information to recreate the structure of an entire filesystem. To do this, we need to have three things: a list of inodes, filesystem metadata for each inode, and a list of hard links (filenames) for each inode. Our system has a hook corresponding to each of these items. Assuming, then, that we can collect provenance for a filesystem starting from the point when it is completely empty, all of this information will appear in the record.

There are two problems with this assumption, however. First, it is impractical. We may connect a USB drive which has been used elsewhere, or we may want to start collecting provenance on an existing, populated filesystem. Second, it is actually impossible to start with an empty filesystem. Without a root inode, which is created by the corresponding `mkfs` program, a filesystem cannot even be mounted. Unfortunately, `mkfs` does this by writing directly to a block device file, which does not generate the expected provenance data.

What we need is a way to bootstrap provenance on a populated filesystem. In order to have a complete record for each file, we must generate a creation event for any pre-existing inodes. We have implemented a utility called `pbang` (for "provenance Big Bang") which does this by traversing the filesystem tree. For each new inode it encounters, it outputs an allocation entry for the inode, a metadata entry containing its attributes, and a link entry containing its filename and directory. For previously encountered inodes, it only outputs a new link entry. All of these entries are written to a file to complete the provenance record. To make a new provenanced filesystem, we create it normally using `mkfs`, then run `pbang` immediately afterward.

5.5 Provenance-Opaque Flag

We noticed a strange behavior in the early prototypes of Hi-Fi: even when the system was completely idle, a continuous stream of provenance data was being generated. Inspection of the provenance record showed that this data described the actions of the provenance handler itself. The handler would call the `read` function to retrieve data from the provenance log, which then triggered the `file_permission` LSM hook. The collector would record this action in the log, where the handler would again read it, triggering `file_permission`, and so on. This created a large amount of "feedback" in the provenance record.

In light of our design goals, this is technically correct behavior. However, it floods the provenance record with data which does not provide any additional insight into the system's operation. One option for solving this problem is to make the handler completely exempt from provenance collection. This, however, has the potential to interfere with our ability to reconstruct the filesystem. If the handler were to create or move a file without generating provenance data, we could no longer accurately reconstruct the filesystem structure from the record. Instead, we make the handler "provenance-opaque," treating it as a black box which only generates provenance data if it makes any significant changes to the filesystem.

The first piece to our solution is informing the LSM which process is the provenance handler. To do this, we leverage the LSM framework's integration with extended filesystem attributes. We identify the provenance handler program by setting an attribute called `security.hifi`. The "security" attribute namespace, which

is reserved for attributes used by security modules, is protected from tampering by malicious users. When the program is executed, the `bprm_check_security` hook examines this property for the value "opaque" and sets a flag in the process's credentials indicating that it should be treated accordingly. In order to allow the handler to create new processes without reintroducing the original problem—for instance, if the handler is a shell script—this flag is propagated to any new credentials that the process creates.

5.6 Socket Provenance

Our modifications to network socket behavior are designed to be both transparent and incrementally deployable. To allow interoperability with existing non-provenanced hosts, we place packet identifiers in the IP Options header field. In order to ensure that every packet sent by our system is marked appropriately, we implement two Netfilter hooks, which process packets at the network layer. The outgoing hook labels each packet with the correct identifier just before it encounters a routing decision, and the incoming hook reads this label just after the receiver decides the packet should be handled locally. Note that even packets sent to the loopback address will encounter both of these hooks.

In designing the log entries for socket provenance, we aim to make the reconstruction of information flows from multiple system logs as simple as possible. When the sender and receiver are on the same host, these entries should behave the same as reads and writes. When they are on different hosts, the only added requirement should be a partial ordering placing each send before all of its corresponding receives. Lamport clocks [12] would satisfy this requirement.

The problem with this is that the `socket_recvmsg` hook, which was designed for access control, executes before a process attempts to receive a message. This may occur before the corresponding `socket_sendmsg` hook is executed. To solve this, we place a `socket_post_recvmsg` hook after the message arrives and before it is returned to the receiver, and we use this hook to generate the entry for receiving a message.

We implement support for TCP and UDP sockets to demonstrate provenance for both connection-mode and connectionless sockets, as well as both stream and message-oriented sockets. Support for the other protocols and pseudo-protocols in the Linux IP stack, such as SCTP, ping, and raw sockets, can be implemented using similar techniques. For example, SCTP is a sequential packet protocol, which has connection-mode and message semantics.

5.6.1 TCP Sockets

TCP and other connection-mode sockets are complicated in that a connection involves three different sockets: the client socket, the listening server socket, and the server socket for an accepted connection. The first two are created in the same way as any other socket on the system: using the `socket` function, which calls the `socket_create` and `socket_post_create` LSM hooks. However, sockets for an accepted connection on the server side are created by a different sequence of events. When a listening socket receives a connection request, it creates a "mini-socket" instead of a full socket to handle the request. If the client completes the handshake, a new child socket is cloned from the listening socket, and the relevant information from the mini-socket (including our IP options) is copied into the child. In terms of LSM hooks, the `inet_conn_request` hook is called when a mini-socket is created, and the `inet_csk_clone` hook is called when it is converted into a full socket. On the client side, the `inet_conn_established` hook is called when the SYN+ACK packet is received from the server.

Our system must treat the TCP handshake with care, since there are two different sockets participating on the server side. We create a unique identifier for the mini-socket in the `inet_conn_request` hook, and this identifier is later copied directly into the child socket. The client must then be certain to remember the correct identifier, namely, the one associated with the child socket. The first packet that the client receives (the SYN+ACK) will carry the IP options from the listening parent socket. To keep this from overriding the child socket, we use the `inet_conn_established` hook to clear the saved identifier so that it is later replaced by the correct one.

5.6.2 UDP Sockets

Since UDP sockets are connectionless, we must use an LSM hook to assign a different identifier to each datagram. In addition, this hook must run in process context, so that we can record the identifier of the process which is sending or receiving. The only existing LSM socket hook with datagram granularity is the `sock_rcv_skb` hook, but it is run as part of an interrupt when a datagram arrives, not in process context. The remaining LSM hooks are placed with socket granularity; therefore, we must place two additional hooks to mediate datagram communication.

The construction and delivery semantics for UDP datagrams are not as straightforward as they may appear at first. An intuitive assumption would be that each datagram is constructed by a single process and received by a single process, but this is not the case. If the file descriptor of the receiving socket is shared between processes, they can all receive the same datagram by using the `MSG_PEEK` flag. In fact, multiple processes can also contribute data when *sending* a single datagram by using the `MSG_MORE` flag or the `UDP_CORK` socket option. Because of this, placing send and receive hooks for UDP is a very subtle task.

Since we consider each datagram an independent entity, the crucial points to mediate are the addition of data to the datagram and the reading of data from it. The Linux IP implementation includes a function which is called from process context to append data to an outgoing socket buffer. This function is called each time a process adds data to a corked datagram, as well as in the normal case where a single process constructs a datagram and immediately sends it. This makes it an ideal candidate for the placement of the send hook, which we call `socket_dgram_append`. Since this hook is placed in network-layer code, it can be applied to any message-oriented protocol and not just UDP.

We also place the receive hook in protocol-agnostic code, for similar flexibility. The core networking code provides a function which retrieves the next datagram from a socket's receive queue. UDP and other message-oriented protocols use this function when receiving, and it is called once for each process that receives a given datagram. This is an ideal location for the message-oriented receive hook, so we place the `socket_dgram_post_recv` hook in this function.

6. EVALUATION

The motivation behind this work is to determine whether whole-system provenance collection can provide useful information in a security context. We demonstrate this in two ways. First, we show that a number of typical malware behaviors appear plainly in a whole-system provenance record. In particular, when malware spreads from one provenanced host to another, we can observe the communication between the infected process on one host and the target process on the other using socket provenance. Second, we demonstrate that the performance overhead of Hi-Fi is small enough that it could be used in practice.

6.1 Recording Malicious Behavior

Our first task is to show that the data collected by Hi-Fi is of sufficient fidelity to be used in a security context. We focus our investigation on detecting the activity of network-borne malware. A typical worm consists of several parts. First, an exploit allows it to execute code on a remote host. This code can be a dropper, which serves to retrieve and execute the desired payload, or it can be the payload itself. A payload can then consist of any number of different actions to perform on an infected system, such as exfiltrating data or installing a backdoor. Finally, the malware spreads to other hosts and begins the cycle again.

For our experiment, we chose to implement a malware generator which would allow us to test different droppers and payloads quickly and safely. The generator is similar in design to the Metasploit Framework [13], in that you can choose an exploit, dropper, and payload to create a custom attack. However, our tool also includes a set of choices for generating malware which automatically spreads from one host to another; this allows us to demonstrate what socket provenance can record about the flow of information between systems. The malware behaviors that we implement and test are drawn from Symantec's technical descriptions of actual Linux malware[23].

To collect provenance data, we prepare three virtual machines on a common subnet, all of which are running Hi-Fi. The attacker generates the malware on machine A and infects machine B by exploiting an insecure network daemon. The malware then spreads automatically from machine B to machine C. For each of the malicious behaviors we wish to test, we generate a corresponding piece of malware on machine A and launch it. Once C has been infected, we retrieve the provenance logs from all three machines for examination.

Each malware behavior that we test appears in some form in the provenance record. In each case, after filtering the log to view only the vulnerable daemon and its descendants, the behavior is clear enough to be found by manual inspection. Below we describe each behavior and how it appears in the provenance record.

6.1.1 Persistence and Stealth

Frequently, the first action a piece of malware takes is to ensure that it will continue to run for as long as possible. In order to persist after the host is restarted, the malware must write itself to disk in such a way that it will be run when the system boots. The most straightforward way to do this on a Linux system is to infect one of the startup scripts run by the `init` process. Our simulated malware has the ability to modify `rc.local`, as the Kaiten trojan does. This shows up clearly in the provenance log:

```
[6fe] write B:/etc/rc.local
```

In this case, the process with provid 0x6fe has modified `rc.local` on B's root filesystem. Persistent malware can also add cron jobs or infect system binaries to ensure that it is executed again after a reboot. Examples of this behavior are found in the Sorso and Adore worms. In our experiment, these behaviors result in similar log entries:

```
[701] write B:/bin/ps
```

for an infected binary, and

```
[710] write B:/var/spool/cron/root.new
[710] link B:/var/spool/cron/root.new to
          B:/var/spool/cron/root
[710] unlink B:/var/spool/cron/root.new
```

for an added cron job.

Some malware is even more clever in its approach to persistence. The Svat virus, for instance, creates a new C header file and places it early in the default include path. By doing this, it affects the code of any program which is subsequently compiled on that machine. We include this behavior in our experiment as well, and it appears simply as:

```
[707] write B:/usr/local/include/stdio.h
```

6.1.2 Remote Control

Once the malware has established itself as a persistent part of the system, the next step is to execute a payload. This commonly includes installing a backdoor which allows the attacker to control the system remotely. The simplest way to do this is to create a new root-level user on the system, which the attacker can then use to log in. Because of the way UNIX-like operating systems store their account databases, this is done by creating a new user with a UID of 0, making it equivalent to the root user. This is what the Zab trojan does, and when we implement this behavior, it is clear to see that the account databases are being modified:

```
[706] link (new) to B:/etc/passwd+
[706] write B:/etc/passwd+
[706] link B:/etc/passwd+ to B:/etc/passwd
[706] unlink B:/etc/passwd+
[706] link (new) to B:/etc/shadow+
[706] write B:/etc/shadow+
[706] link B:/etc/shadow+ to B:/etc/shadow
[706] unlink B:/etc/shadow+
```

A similar backdoor technique is to open a port which listens for connections and provides the attacker with a remote shell. This approach is used by many pieces of malware, including the Plupii and Millen worms. Our experiment shows that the provenance record includes the shell's network communication as well as the attacker's activity:

```
[744] exec B:/bin/bash -i
[744] socksend B:173
[744] sockrecv unknown
[744] socksend B:173
[751] exec B:/bin/cat /etc/shadow
[751] read B:/etc/shadow
[751] socksend B:173
[744] socksend B:173
[744] sockrecv unknown
[744] socksend B:173
[744] link (new) to B:/testfile
[744] write B:/testfile
```

Here, the attacker uses the remote shell to view `/etc/shadow` and to write a new file in the root directory. Since the attacker's system is unlikely to be running a trusted instance of Hi-Fi, we see "unknown" socket entries, which indicate data received from an unprovenanced host. Remote shells can also be implemented as "reverse shells," which connect from the infected host back to the attacker. Our tests on a reverse shell, such as the one in the Jac.8759 virus, show results identical to a normal shell.

6.1.3 Exfiltration

Another common payload activity is data exfiltration, where the malware reads information from a file containing password hashes, credit card numbers, or other sensitive information and sends this information to the attacker. Our simulation for this behavior reads the `/etc/shadow` file and forwards it in one of two ways. In the first test, we upload the file to a web server using HTTP, and in the second, we write it directly to a remote port. Both methods result in the same log entries:

```
[85f] read B:/etc/shadow
[85f] socksend B:1ae
```

Emailing the information to the attacker, as is done by the Adore worm, would create a similar record.

6.1.4 Spread

Our experiment also models three different mechanisms used by malware to spread to newly infected hosts. The first and simplest is used when the entire payload can be sent using the initial exploit. In this case, there does not need to be a separate dropper, and the resulting provenance log is the following (indentation is used to distinguish the two hosts):

```
[807] read A:/home/evil/payload
[807] socksend A:153
    [684] sockrecv A:153
    [684] write B:/tmp/payload
```

The payload is then executed, and the malicious behavior it implements appears in subsequent log entries.

Another mechanism, used by the Plupii and Sorso worms, is to fetch the payload from a remote web server. We assume the web server is unprovenanced, so the log once again contains "unknown" entries:

```
[7ff] read A:/home/evil/dropper
[7ff] socksend A:15b
    [685] sockrecv A:15b
    [685] write B:/tmp/dropper
    [6ef] socksend B:149
    [6ef] sockrecv unknown
    [6ef] write B:/tmp/payload
```

If the web server were a provenanced host, this log would contain host and socket IDs in the `sockrecv` entry corresponding to a `socksend` on the server.

Finally, to illustrate the spread of malware across several hosts, we tested a "relay" dropper which uses a randomly-chosen port to transfer the payload from each infected host to the next. The combined log of our three hosts shows this process:

```
[83f] read A:/home/evil/dropper
[83f] socksend A:159
    [691] sockrecv A:159
    [691] write B:/tmp/dropper
    [6f5] exec B:/tmp/dropper
[844] read A:/home/evil/payload
[844] socksend A:15b
    [6fc] sockrecv A:15b
    [6fc] write B:/tmp/payload
    [74e] read B:/tmp/dropper
    [74e] socksend B:169
        [682] sockrecv B:169
        [682] write C:/tmp/dropper
        [6e6] exec C:/tmp/dropper
    [750] read B:/tmp/payload
    [750] socksend B:16b
        [6ed] sockrecv B:16b
        [6ed] write C:/tmp/payload
```

Here we can see the attacker transferring both the dropper and the payload to the first victim using two different sockets. This victim then sends the dropper and the payload to the next host in the same fashion.

6.1.5 Full Simulation

For a comprehensive test, we use our tool to implement a full simulation of the Linux Adore worm according to Symantec's description. Our provenance record captures the entire life cycle of the worm:

System call	Baseline	With Hi-Fi	Overhead
open	13.8	13.8	0.0%
close	10.6	10.7	1.0%
read	13.7	14.6	6.2%
write	21.4	21.3	-0.2%
creat	24.1	24.4	1.1%
rename	19.8	20.0	0.9%
unlink	36.4	36.7	0.7%
clone	74.6	74.0	-0.7%
execve	150.3	155.1	3.2%

Table 2: Mean execution time for system calls (µs)

- The compromised daemon downloading and extracting the payload tarball
- Execution of `start.sh`, which activates the payload
- Replacement of the `ps` binary with a trojaned version, and copying the original `ps` to `/usr/bin/adore`
- Installation of a cron job which kills the worm
- Replacement of `klogd` with a backdoor shell
- Emailing of the `/etc/shadow` file, process list, and network information to the attacker
- Infection of the next victim

We also successfully capture a sample backdoor session, in which the attacker views a user's command-line history and downloads an updated payload.

6.2 Performance

In addition to showing that Hi-Fi records malicious activity, we also wish to show that it does so without significantly degrading system performance. To this end, we benchmark a system running a stock Arch Linux kernel (version 3.2.13), then benchmark the same system with Hi-Fi compiled in. Our test system has two 2.30-GHz quad-core AMD Opteron processors, 16GB of RAM, and two 73GB hard disks in a RAID 0 array.

We first evaluate performance overhead at the system-call level using microbenchmarks. LMbench is frequently used for Linux microbenchmarks, but our initial results from this tool were inconsistent. Instead, we create a small program which exercises the major file and process operations. We then use the `strace` utility to measure the time spent in various system calls over a large number of executions of this program. The results of these benchmarks are summarized in Table 2. For the system calls measured, the overhead is at most 6.2%, with most calls within 1% of the baseline.

To demonstrate the overall impact on system performance, we also run two macrobenchmarks customarily used in provenance system evaluation: a Linux kernel build, which evaluates a typical combination of process execution and file manipulation; and PostMark [11], which specifically stresses filesystem and disk transactions. We generate statistics from multiple executions of each benchmark using the Phoronix Test Suite utility [17]. With an unmodified kernel, our test system takes an average of 107 seconds to run the kernel-build benchmark. With Hi-Fi, this increases to 110 seconds, showing an overhead of only 2.8%. Performance on disk-heavy operations is unchanged, as PostMark achieves 2,083 transactions per second in both cases.

7. CONCLUSION

We have presented Hi-Fi, a system which applies the reference monitor concept to collect a high-fidelity provenance record suitable for security applications. We show that this record can be used to observe the behavior of malware, not only within a single host, but also across multiple provenanced hosts. Furthermore, we demonstrate that our implementation imposes less than 3% overhead on representative workloads and a similarly small overhead in system-call microbenchmarks.

We believe that Hi-Fi will provide a solid platform for future provenance research. For example, we do not explore options for working with provenance data after it is collected, but the modular design of Hi-Fi will make it simple to evaluate many different approaches to processing, storage, and querying. We have shown that complete system-level and socket provenance can provide deep insight into the design, performance, and security of systems and networks, and we believe that many other significant discoveries are yet to be made in this area.

Acknowledgements

This work is supported by the National Science Foundation under awards HECURA-0937944 and CNS-1118046.

8. REFERENCES

[1] J. P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, AFSC, Hanscom AFB, Bedford, MA, Oct. 1972. AD-758 206, ESD/AFSC.

[2] U. Braun, S. Garfinkel, D. Holland, K. Muniswamy-Reddy, and M. Seltzer. Issues in automatic provenance collection. In *Proceedings of the 2006 International Provenance and Annotation Workshop*, pages 171–183, 2006.

[3] A. Edwards, T. Jaeger, and X. Zhang. Runtime verification of authorization hook placement for the Linux Security Modules framework. In V. Atluri, editor, *ACM Conference on Computer and Communications Security*, pages 225–234. ACM, 2002.

[4] Filesystem in userspace. `http://fuse.sourceforge.net`.

[5] V. Ganapathy, T. Jaeger, and S. Jha. Automatic placement of authorization hooks in the Linux Security Modules framework. In V. Atluri, C. Meadows, and A. Juels, editors, *ACM Conference on Computer and Communications Security*, pages 330–339. ACM, 2005.

[6] D. Garg, L. Jia, and A. Datta. Policy auditing over incomplete logs: theory, implementation and applications. In *Proceedings of the 18th ACM conference on Computer and Communications Security*, CCS '11, pages 151–162, New York, NY, USA, 2011. ACM.

[7] A. Goel, K. Farhadi, K. Po, and W.-c. Feng. Reconstructing system state for intrusion analysis. *SIGOPS Oper. Syst. Rev.*, 42(3):21–28, Apr. 2008.

[8] A. Goel, W.-C. Feng, D. Maier, and J. Walpole. Forensix: a robust, high-performance reconstruction system. In *Distributed Computing Systems Workshops, 2005. 25th IEEE International Conference on*, pages 155–162, june 2005.

[9] R. Ikeda and J. Widom. Panda: A system for provenance and data. *IEEE Data Engineering Bulletin*, September 2010.

[10] S. N. Jones, C. R. Strong, D. D. Long, and E. L. Miller. Tracking emigrant data via transient provenance. In *Third Workshop on the Theory and Practice of Provenance*. USENIX, June 2011.

[11] J. Katcher. Postmark: a new file system benchmark. Network Appliance Tech Report TR3022, Oct. 1997.

[12] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[13] Metasploit Project. `http://www.metasploit.com`.

[14] L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. T. Groth, N. Kwasnikowska, S. Miles, P. Missier, J. Myers, B. Plale, Y. Simmhan, E. G. Stephan, and J. V. den Bussche. The Open Provenance Model core specification (v1.1). *Future Generation Comp. Syst.*, 27(6):743–756, 2011.

[15] K. Muniswamy-Reddy, J. Barillari, U. Braun, D. Holland, D. Maclean, M. Seltzer, and S. Holland. Layering in provenance-aware storage systems. In *Proceedings of the 2009 USENIX Annual Technical Conference, San Diego, CA*, 2009.

[16] K. Muniswamy-Reddy and D. Holland. Causality-based versioning. *ACM Transactions on Storage (TOS)*, 5(4):13, 2009.

[17] Phoronix Test Suite. `http://phoronix-test-suite.com`.

[18] C. F. Reilly and J. F. Naughton. Transparently gathering provenance with provenance aware Condor. In *First workshop on theory and practice of provenance*, TAPP'09, pages 13:1–13:10, Berkeley, CA, USA, 2009. USENIX Association.

[19] C. Sar and P. Cao. Lineage file system. *Online at http://crypto.stanford.edu/cao/lineage.html*, 2005.

[20] R. Sion. Strong worm. In *Proceedings of the 2008 The 28th International Conference on Distributed Computing Systems*, 2008.

[21] R. P. Spillane, R. Sears, C. Yalamanchili, S. Gaikwad, M. Chinni, and E. Zadok. Story Book: An efficient extensible provenance framework. In J. Cheney, editor, *Workshop on the Theory and Practice of Provenance*. USENIX, 2009.

[22] S. Sundararaman, G. Sivathanu, and E. Zadok. Selective versioning in a secure disk system. In *Proceedings of the 17th conference on Security symposium*, 2008.

[23] Symantec Security Response. `http://www.symantec.com/security_response`.

[24] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou. AutoISES: Automatically inferring security specification and detecting violations. In P. C. van Oorschot, editor, *USENIX Security Symposium*, pages 379–394. USENIX Association, July 2008.

[25] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *CIDR*, pages 262–276, 2005.

[26] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux Security Modules: General security support for the Linux kernel. In USENIX, editor, *Proceedings of the 11th USENIX Security Symposium 2002, August 5–9, 2002, San Francisco, CA*, pages 17–31. USENIX, 2002.

[27] T. Zanussi, K. Yaghmour, R. Wisniewski, R. Moore, and M. Dagenais. relayfs: An efficient unified approach for transmitting data from kernel to user space. In *Proceedings of the 2003 Linux Symposium, Ottawa, ON, Canada*, pages 494–506, July 2003.

[28] X. Zhang, A. Edwards, and T. Jaeger. Using CQUAL for static analysis of authorization hook placement. In D. Boneh, editor, *USENIX Security Symposium*, pages 33–48. USENIX, 2002.

Transforming Commodity Security Policies to Enforce Clark-Wilson Integrity

Divya Muthukumaran
Pennsylvania State University
muthukum@cse.psu.edu

Sandra Rueda
Universidad de los Andes
sarueda@uniandes.edu.co

Nirupama Talele
Pennsylvania State University
nrt123@psu.edu

Hayawardh Vijayakumar
Pennsylvania State University
hvijay@cse.psu.edu

Jason Teutsch
Pennsylvania State University
teutsch@cse.psu.edu

Trent Jaeger
Pennsylvania State University
tjaeger@cse.psu.edu

ABSTRACT

Modern distributed systems are composed from several off-the-shelf components, including operating systems, virtualization infrastructure, and application packages, upon which some custom application software (e.g., web application) is often deployed. While several commodity systems now include mandatory access control (MAC) enforcement to protect the individual components, the complexity of such MAC policies and the myriad of possible interactions among individual hosts in distributed systems makes it difficult to identify the attack paths available to adversaries. As a result, security practitioners *react* to vulnerabilities as adversaries uncover them, rather than *proactively protecting* the system's data integrity. In this paper, we develop a mostly-automated method to transform a set of commodity MAC policies into a system-wide policy that proactively protects system integrity, approximating the Clark-Wilson integrity model. The method uses the insights from the Clark-Wilson model, which requires integrity verification of security-critical data and mediation at program entrypoints, to extend existing MAC policies with the proactive mediation necessary to protect system integrity. We demonstrate the practicality of producing Clark-Wilson policies for distributed systems on a web application running on virtualized Ubuntu SELinux hosts, where our method finds: (1) that only 27 additional entrypoint mediators are sufficient to mediate the threats of remote adversaries over the entire distributed system and (2) and only 20 additional local threats require mediation to approximate Clark-Wilson integrity comprehensively. As a result, available security policies can be used as a foundation for proactive integrity protection from both local and remote threats.

1. INTRODUCTION

A large fraction of modern computation is now deployed in distributed systems consisting of several, independently-developed software components. For example, web applications (e.g., a LAMP software bundle) consist of: (1) an operating system distribution and its system services (e.g., Linux); (2) a web server (e.g., Apache); (3) a database and other backend software (e.g., MySQL), and (4) custom server code (e.g., written in PHP) to which web clients connect to perform a variety of critical applications. Each of these components face their own threats and connecting them together into a distributed system only increases the avenues that adversaries can leverage to compromise the system.

Computing system compromises occur because *data integrity* is not managed effectively. Adversaries use the open accessibility to many distributed systems to attempt attacks ranging from malformed network packets to embedded executable content to imported files containing malware. One mechanism introduced into commodity systems to combat such attacks is *mandatory access control* (MAC) [33, 49, 51, 54, 27]. MAC enforcement limits processes to program-specific permissions to protect the kernel's integrity, even from some root processes[1]. MAC enforcement is now available in virtual machine monitors [9, 42, 22] (VMMs) and user-level programs [29], in addition to operating systems, enabling such control throughout the system. Further, such MAC enforcement is now integrated with network access control [26, 17, 32], presenting an opportunity for comprehensive access control in commercial deployments. However, the addition of all this enforcement does not seem to be changing the dynamics of security management. Preventing compromises is still a reactive task, fixing vulnerabilities as adversaries identify them.

We find that the current approach to securing systems using MAC enforcement forces administrators to be reactive. Commodity MAC policies often consist of many complex policy rules, so configuring MAC policies is now a task undertaken only by experts. As a result, administrators use the default MAC policies provided in OS distributions to protect their distributed systems blindly. However, commodity system MAC policies are designed based on the expected functionality required by processes, resulting in *least privilege* enforcement [43]. Unfortunately, almost every process in a commodity system is accessible to adversaries, even in its least privilege operation [53], so current commodity MAC

[1]A small set of programs are authorized to modify MAC policies in practice, but not all root processes as was the case previously.

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

policies do not protect process integrity comprehensively. While researchers have developed analysis tools to detect potential problems in commodity MAC policies [18, 52, 45, 55, 7], determining the resolutions to such problems is still a complex manual task, requiring MAC policy expertise. The result is that it is not practical for administrators to foresee all the possible attack paths available to adversaries, causing them to react to adversary exploits.

Instead, our goal is to generate MAC policies that protect system-wide integrity from the available security policies. Our insights are motivated by the Clark-Wilson integrity model [8]. The Clark-Wilson integrity model defines strict requirements for protecting integrity where *integrity verification procedures* (IVPs) validate high integrity data, only certified *transformation procedures* (TPs) modify high integrity data, and TPs protect themselves by upgrading or discarding any low integrity inputs they may receive. While MAC policies define the flows to and from processes, they fail to identify the data whose integrity is critical to those processes (e.g., require IVPs) or which process entrypoints[2] must protect themselves from adversary access. However, by using the available MAC policies, we find that we can construct an information flow problem that enables computation of placements of integrity verification and entrypoint mediation necessary to resolve all information flow integrity errors in the system.

In this paper, we develop a mostly-automated approach to compute the set of integrity verification and entrypoint mediation sufficient to protect information flow integrity in distributed systems. First, this method constructs system-wide data flow graphs from available MAC policies and "connection" policies (e.g., firewall policies) that describe how individual software components communicate in distributed systems. Second, we create information flow problems by adding integrity semantics to nodes in the data flow graph semi-automatically. Third, we compute a minimal placement of integrity verification and entrypoint mediation necessary to resolve all information flow errors automatically. Using this placement, we can produce a system-wide data integrity policy that approximates Clark-Wilson integrity as described above. We have found [41] that the resultant policies are equivalent to Decentralized Information Flow Control policies [20], possibly opening the way to leverage information flow-based enforcement mechanisms for commodity system deployments.

This paper makes the following contributions. First, we define an information flow problem whose data flow graph represents an hierarchical system of software components with independent entry and exitpoints. This enables us to model information flow system-wide and identify the need for mediation at program entrypoints. Second, we design a method to solve such problems using graph cuts, where the locations of possible cuts are constrained by the integrity of program packages. Third, we demonstrate our method on a custom web application running on virtualized Ubuntu SELinux hosts, which finds that only 27 additional entrypoint mediators and 20 integrity verification procedures are necessary to provide information flow integrity approximating Clark-Wilson integrity. This is the first approach to produce system-wide access control policies targeted to classical integrity in a mostly-automated fashion.

[2]A program entrypoint is a program instruction that receives input from the operating system, such as the caller of the library function that invoke a system call.

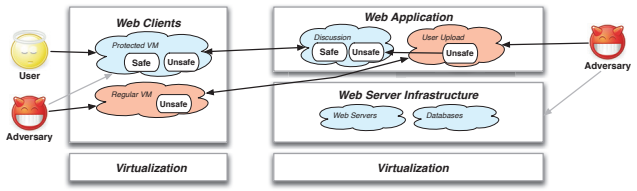


Figure 1: Example web application consisting of a web server with three web application components that handle data of different integrity levels on a standard web infrastructure. The web client is also divided into two VMs for regular and protected operation.

The remainder of the paper is structured as follows. In Section 2, we motivate the problem of deploying a secure web application using commodity MAC policies. In Section 3, we motivate the use of Clark-Wilson integrity as a guide for resolving information flow errors. In Section 4, we describe how to construct and solve the information flow problems necessary to compute an approximation of a Clark-Wilson integrity policy system-wide. In Section 5, we describe the tool implementation and evaluation results. In Sections 6 and 7, we detail related work and conclude the paper.

2. PROBLEM DEFINITION

2.1 Motivation

When administrators deploy distributed applications, one problem that they must address is whether that deployment protects the application's data integrity end-to-end. As an example, consider a web application that enables collaborative decision-support for groups of users shown in Figure 1. Such applications consist of two main tasks: data gathering (upload) and data analysis for decision-making (discussion). First, data is often gathered from external, perhaps untrustworthy, sources by some clients for others to evaluate. In this example, users use the regular web clients to gather data for upload to the *upload* web application component. Then, users use the protected client to examine this data and perform decision-making (with other clients) using the *discussion* web application component. To do this, the discussion web application component produces two streams of content, one (potentially) untrusted stream from external sources (from upload) and a second, high integrity stream from collaborative discussions of trusted sources, where both streams may be displayed in a manner suitable for users to distinguish one content stream from the other [13].

While there are multitude of threats that are possible against such applications, we identify three specific threats that represent the different types of problems we aim to address. First, the web infrastructure (e.g., LAMP software bundle including web server, database, etc.) may be threatened by remote adversaries. While the web infrastructure often consists of mature software that has been hardened against many threats as the result of years of penetrate-and-patch, new vulnerabilities continue to be discovered. Second, we have to be careful that a web application deployment does not introduce new attack paths for itself and the infrastructure. The web application components are gener-

ally custom software implemented for the specific web application, so they are not necessarily hardened to the threats that they may face. Remote threats that compromise one web application component may be used as a stepping to attack other web applications and the web infrastructure. Third, local threats may also be used to attack the web application. Some vulnerabilities are caused by trusting files that may have been supplied by adversaries by tricking users or compromised programs to import them. For example, a user may be tricked into importing malicious libaries to their home directory, which may lead to a untrusted search path vulnerability.

As a result, it is difficult to deploy web applications that protect data integrity end-to-end. Administrators currently focus their efforts on management of the web infrastructure, which consists of mature programs, runs on hosts they manage directly, and is well-supported by vulnerability reporting. Administrators assume that web infrastructure software prevents known threats, and they can track vulnerability reports to determine whether new vulnerabilities will require upgrades or software package changes. This is not the case for web application components and web client software, however. Such programs may be new, be managed at least partially by end users, and be sufficiently ad hoc that it is difficult to determine how a new vulnerability may affect them. That is, the reactive approach may not be effective because only their configuration may have the vulnerability and the method of reaction may differ between web application configurations. To address this limitation, we aim to develop a proactive approach to protecting data integrity.

2.2 Proactive Integrity

Designing for integrity traditionally requires finding solutions to an information flow problem that satisfies an integrity policy. For example, Biba integrity requires that no process receive any information flow (read or execute) containing data whose integrity level is lower than that of the process [5]. The information flow problem can be expressed using the model below:

DEFINITION 2.1. *Let $\mathcal{I}$ be an* information flow problem*, $\mathcal{I} = (\mathcal{G}, \mathcal{L}, \mathcal{M})$, to find whether whether a data flow graph G with a level mapping function M for a lattice $\mathcal{L}$ contains any information flow errors, where:*

1. *A* data flow graph $G = (V, E)$ *consists of a set of nodes V connected by a set of directed edges E.*
2. *There is a* lattice $\mathcal{L} = \{L, \preceq\}$*. For any two levels $l_i, l_j \in L$, $l_i \preceq l_j$ means that l_i 'can flow to' l_j.*
3. *There is a* level mapping function $M : V \rightarrow \mathcal{P}^L$ *where $\mathcal{P}^L$ is the power set of L (i.e., each node is mapped either to a set of levels in L or to $\emptyset$).*
4. *The lattice imposes security constraints on the information flows enabled by the data flow graph. Each pair $u, v \in V$ s.t. $[u \hookrightarrow_G v \wedge (\exists l_u \in M(u), l_v \in M(v). l_u \not\preceq_{\mathcal{L}} l_v)]$, where $\hookrightarrow_G$ means there is a path from u to v in G, represents an* information flow error*.*

It has been shown that information flow errors in programs [28] and MAC policies [18, 45, 52] can be found automatically using such a model. For integrity, the lattice would represent an integrity policy, and the level mapping function would map levels in that integrity lattice to subjects and objects in the system.

Through the addition of a variety of security mechanisms over the last 10 years, administrators can now make a number of choices to protect the integrity of their deployments proactively, but the sum of these measures fall short of satisfying information flow integrity. For example, administrators can: (1) configure firewall policies, which define the possible data flows among hosts (including virtual hosts); (2) choose OS distributions with mandatory access control (MAC) policies (supported in several commodity systems [51, 33, 31, 27, 54]), which define the possible data flows among processes running on that OS; and (3) choose software packages to run on their hosts, which defines the possible data flows within programs. The problem is that many of these authorized data flows allow adversaries to access integrity-critical data and processing. The reason for this is that such policies are designed to enforce *least privilege* [43], where programs are only limited to the permissions required for them to function properly. However, in a runtime analysis study, we found that nearly every program is designed to receive some adversary-supplied data [53], which can lead to vulnerabilities when such untrusted data is accessed in unexpected ways or has unexpected values.

Researchers have long recognized this problem, but thusfar the solutions proposed involve significant, manual effort. In general, information flow errors may be resolved by *mediators*, which ensure that the runtime behavior of the system is consistent with its security requirements (i.e., integrity lattice). Mediators may be implemented as entire processes, such as guards, or as individual program statements, such as endorsers in security-typed languages [29]. In the information flow problem, a mediator associates an edge $(u, v) \in E$ with an integrity level to which data is raised $l \in L$ when transmitted on the edge by node u.

As system functionality often violates information flow integrity, the placement of mediators necessary to resolve such errors is an important and difficult problem. Recently, researchers have proposed a variety of approaches to enforce information flow integrity focusing on system abstractions [21, 47, 50, 20, 56, 57]. All of the above approaches require that administrators replace the commodity system policies with a new information flow policy that includes mediators. Only Practical Proactive Integrity [50] (PPI) provides some automated support for producing integrity policies from existing MAC policies, but it uses simple, two-level lattices and requires administrators to determine whether a process can handle all integrity threats, which is a difficult and error-prone task. To evaluate such risks more precisely, researchers have realized that it is important to identify and defend those program entrypoints accessible to adversaries[3], called the program's *attack surface* [16, 23]. Some of the information flow integrity enforcement mechanisms above reason about integrity based on the program entrypoints [47, 20]. See Section 6 for further details.

Rather than requiring administrators to specify a new information flow policies manually, we argue that such policies, including the mediators necessary to prevent information flow errors, can be computed automatically. Our goal is to use the system's available security policies to produce a system-wide policy with the minimal mediation necessary

[3]A program entrypoint is a program instruction that receives input from the operating system.

to resolve the system's information flow errors as defined in Definition 2.1. We are motivated by prior work that demonstrated that a placement of mediators that resolves all information flow errors is equivalent to a cut of error paths in the data flow graph [37, 19]. However, to make this idea practical, we explore how to produce an information flow policy for the example web application that resolves its local and remote threats. As a result, we find several additional challenges must be addressed, such as finding practical mediator options and producing cuts for general lattice policies. Thus, this work is the first to produce mediator placements necessary to resolve information flow errors system-wide. In a separate technical report [41], we prove that solutions to the information flow problem above are equivalent to legal Decentralized Information Flow Control policies in the Flume model [20], possibly opening the way to leverage information flow-based enforcement for commodity system deployments.

3. SOLUTION APPROACH

Our approach is motivated by two insights that enable the solution of information flow problems from the available security policies in commodity system deployments.

Clark-Wilson Mediators. As a guide for where to place mediators, we turn to the Clark-Wilson integrity model [8], which consists of rules that define the high integrity operation of a system. Of particular interest are the rules that define how high integrity data is processed securely, where: (1) high integrity data (CDIs) must satisfy *integrity verification procedures* (IVPs) (Clark-Wilson rule C1); (2) only approved programs called *transformation procedures* (TPs) may modify high integrity data (C2, E1); and (3) TPs may only receive low integrity data (UDIs) if that data is upgraded or discarded (C5). That is, TPs protect data integrity, but they require IVPs to validate that the data an application depends upon is high integrity and TPs must be capable of mediating low integrity inputs to upgrade or discard such data.

Clark-Wilson identifies two types of mediators: IVPs and the TP program entrypoints. In practice, we find that IVPs are useful for mediating the integrity of files imported into the system. In our example, the protected web client may include restrictions on the files introduced into the user's directory, which acts as a mediator to protect the processes that use such files. Examples of possible IVP mediation include signed package files, binding files to their values at installation [46], and policy-sealed data [44]. If the network inputs and/or local files cannot be validated by IVPs, then entrypoint mediation is placed to protect processes. In general, entrypoint mediation is application-specific, although researchers have proposed general methods to prevent some attacks, such as input handling libraries [39] and safe name resolution [6]. In this paper, we produce an information flow policy that places mediators sufficient to protect the system's data integrity, but not the mediation code. Providing effective mediation code remains an open research issue for all the current information flow enforcement approaches [20, 29], but we are the first to design a method for system-wide placement of mediation.

Choosing Practical Mediation. Computing mediation placements solely from the information flow problem alone ignores some practical considerations. First, several programs, particularly mature ones, have already been hardened by the reactive approach described earlier. We want to reuse those mediators in choosing placements. Second, mediator placement locations may be limited in their ability to solve information flow problems. For example, the web application may need to filter untrusted input data, but it cannot be trusted to protect the kernel's integrity. In past work, these considerations are left as manual tasks.

To address these practical problems, we compute: (1) the mediator placements for infrastructure components to enable their reuse in deployments and (2) the constraints to limit the maximum integrity level that each mediator may endorse. First, we break the task of computing mediation into two steps: we first compute the mediators required for a default install of the infrastructure VMs to estimate the mediation that is necessary for any deployment, and then we compute the additional mediators required for the deployed application at large on this infrastructure. As described above, the movement toward pre-configured VMs encourages their reuse, so we further encourage the reuse of what should be their fundamental mediation, required in all deployments. Second, we compute constraints on the maximum integrity level for each possible mediator location for input to the mediator placement method. Producing these constraints is based on prior work that creates mutual-integrity partitions of labels from MAC policies [53].

These insights can produce a variety of positive effects on computing mediator placements system-wide. Using Clark-Wilson integrity as a guide enables mediation placement for programs that do not enforce MAC policies explicitly, reduces the number of mediation locations to consider significantly, and provides defense for both local and remote threats. By estimating infrastructure mediation and limiting the scope of mediation, we distinguish between the mediation sufficient to protect infrastructure in general and that additional mediation required when those infrastructure components are combined with application software and data and connected into a distributed system. By constraining the integrity levels to which mediators may raise data, we limit the number of programs that can endorse data at each integrity level. We evaluate each of these possible effects in Section 5.

Assumptions. The key assumption in this work is that the programs, operating systems, and firewalls that enforce MAC policies do so correctly. This is a significant assumption given the size and complexity of such software, but it is the standard assumption in modern computing systems. Specifically, we assume that the programs, operating systems, and firewalls satisfy the *reference monitor concept* [2], which requires that a reference validation mechanism (i.e., MAC enforcement) must "must always be invoked" upon a security-sensitive operation, "must be tamperproof," and must be "small enough to be subject to analysis and tests, the completeness of which can be assured," which implies correctness. The reference monitor concept is certainly the goal of these MAC-enforcing commodity systems, even if they do not meet the letter of these requirements.

4. DESIGN

To produce a mediator placement system-wide, we need to construct a system-wide information flow problem (see Definition 2.1) and compute a placement that resolves all the information flow errors in that problem. Given available security policies, we claim that building system-wide infor-

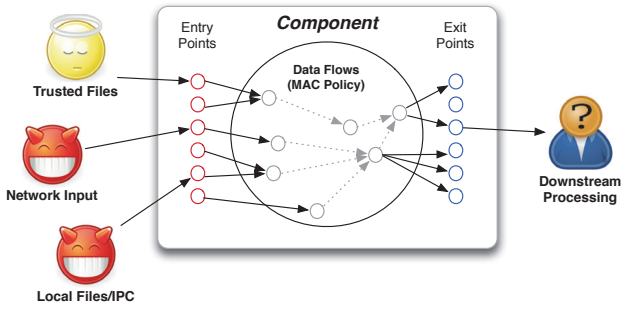


Figure 2: Each software component consists of entry and exit nodes. When the component enforces its own MAC policy then the entry and exit nodes are connected to nodes in the MAC policy.

mation flow problems can be largely automated, resulting a similar effort as configuring information flow problems for single entities (programs or MAC policies). We design an automated method to produce mediator placements from such problems that is an extension of the basic graph cut idea [37, 19] to address general lattices and constrained mediators.

4.1 Building Information Flow Problems

Constructing information flow problems cannot be completely automated because we cannot predict application-specific requirements. Nonetheless, we can greatly reduce the effort necessary to configure such problems, even in complex, distributed environments, such that the task of configuring a system-wide information flow problem is comparable to configuring information flow problems for one entity (e.g., as expected by prior analyses for programs [29] or systems [52]). In this section, we describe the two main insights that guide the construction of information flow problems.

Connection Policies. A modern system deployment now consists of several reference monitors (e.g., firewall, OS, program) independently enforcing access control policies. When enforcing mandatory access control, these policies represent the possible data flows among subjects and objects governed by their respective reference monitors. However, because security policies for each reference monitor are specified independently, the flows among subjects and objects belonging to different reference monitors are ambiguous. For example, while firewall policies limit how adversaries may access the host by port, the specific host processes using those ports are not identified explicitly. Researchers have addressed this particular ambiguity by introducing labeled networking [4], for which there are several implementations in Linux alone [17, 32, 26]. The problem is that administrators must then understand the policies (and implications therein) of each reference monitor necessary to connect the data flows to produce a system-wide data flow graph.

Instead, we find that such connections are either well-known or can be derived automatically, so administrator specification is unnecessary. In the web application, many subjects can be inferred by the use of privileged ports. Also, given the emergence of purpose-specific VMs, the subjects that can possibly use network resources can be easily identified (e.g., included with its specification). For the web application example, the use of unprivileged ports by the browser VMs must be limited to browser processing. A similar problem occurs when connecting program entrypoints to

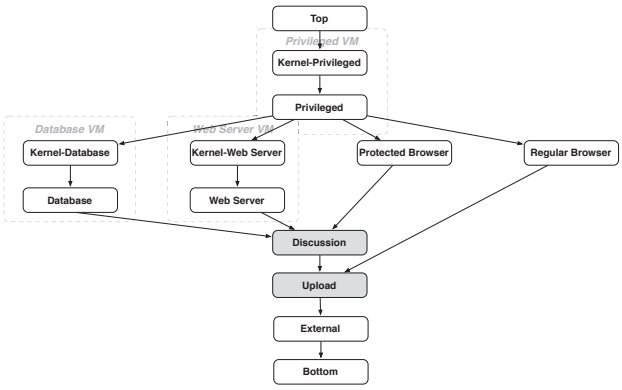


Figure 3: Integrity lattice for the web application. The infrastructure VM levels are highlighted and the application levels are shaded.

the subject and object labels in the MAC policy accessed by those entrypoints. In this case, we use runtime analysis to collect these data flows in a manner analogous to the construction of MAC policies from the permissions programs request [38, 31]. Thus, to construct a system-wide data flow graph for the web application, no administrator specification is necessary, if the distributors of purpose-specific VMs include connections between the network and VM processes (manual) and between program entrypoints and MAC labels (runtime analysis).

Using this information, we automatically construct a data flow graph consisting of components shown in Figure 2. Each component represents an operating system or program by its set of entrypoints, a module containing internal authorized data flows, and exitpoints. For operating systems, the MAC policy they enforce creates a subgraph within the module. For programs, we treat the program internals as a single node. If a program is information-flow aware, it can be represented in the same manner as a MAC enforcing operating system. The system is represented by a hierarchical graph of these components (we use the model defined by Alur [1]), where the network is the highest level, followed by hosts (operating systems) and programs.

Standard Infrastructure Mappings. In order to construct an information flow problem, administrators must produce an integrity lattice and level mapping function to the appropriate subjects and objects in each of the security policies in the distributed system. This task is difficult and error-prone. Researchers typically address this problem by limiting the scope of the information flow problem to a single reference monitor, where an expert can define the expected mapping for the system entities or program variables.

To simplify this problem, we reuse the integrity lattice and level mapping functions defined for each of the infrastructure components when building the system-wide information flow problem to limit the scope of effort required by administrators. We have defined an integrity lattice and level mappings for the web server, database, and privileged VM, and have been able to reuse them unchanged for various web application deployments. Since infrastructure aims to support a wide variety of application deployments essentially unchanged, we expect that reuse for other deployments will be likely. As a result, administrators will only have to specify integrity requirements for their application components.

For the web application, we only need to identify the two types of application data for discussion and external streams of data. Figure 3 shows the integrity lattice for the web application, distinguishing infrastructure levels from those in the application.

4.2 Computing Minimal Mediation

Researchers had the insight that placing a mediator to resolve information flow errors for a lattice policy containing two levels l_i and l_j is tantamount to generating an edge cut[4] of the data flow graph with the nodes mapped to l_i as the sources and the nodes mapped to l_j as the sinks [37, 19]. This property is called *Cut-Mediation Equivalence*. In practice, general lattices policies must be enforced, as shown in Figure 3, so we customize the solution to account for such policies, restricting components that have limited mediation abilities (i.e., only mediate for some, but not all, errors).

Given a directed graph G=(V, E) and a lattice $\mathcal{L}$=$\{L, \preceq\}$, there is a *cut problem* when there is an information flow error between two nodes with the level mapping function M. For general lattices, we must ensure that only information flows authorized by the lattice are possible in the information flow problem. In the web application, we must both protect the application from untrusted inputs, the operating systems in each VM from its applications, and virtualization infrastructure from the guest VMs. As a result, a general lattice creates a set of cut problems to solve, as many as one for each pair of integrity levels in the lattice.

DEFINITION 4.1. *A* cut problem set*, C is defined as $C = \{(l_i, l_j) | l_i, l_j \in L$, where $l_i \not\preceq l_j$, $[\exists\, u, v \in G\ .u \hookrightarrow_G v \wedge l_i \in M(u) \wedge l_j \in M(v)]\}$, consists of such pairs of integrity levels l_i and l_j. The problem of finding the minimal cut for a cut problem set is called the* multiway cut problem*.*

Researchers have shown that the multiway cut problem is NP-Hard for directed graphs [12]. In the context of security lattices, researchers previously suggested a simple greedy solution to the problem that returns the *union* of the solutions for each individual cut problem [19].

To improve on the simple greedy approach, we use the insight that classical integrity encourages processes to upgrade input integrity to their level [8]. The semantics of mediation imply that a node that is picked as a mediator (a node in the graph cut) solution, will raise the level of the incoming data in order to resolve the information flow error. We also note that each node has a limit regarding how high it may upgrade any data; we call this the *maxraiselevel* of each node. Therefore, the graph-cut procedure produces *mediators*, $R \subseteq E \times L$, where each mediator upgrades the integrity of the output data on the edge (u, v) to an $l \in L$, which is at most the node u's *maxraiselevel*. A node's *maxraiselevel* is determined by its program and the levels mapped to it. Using prior work [53], we identify equivalence classes of labels in the MAC policy whose programs must mutually trust one another's integrity. Each node's *maxraiselevel* is the greatest lower bound of any mapping to a label in its integrity equivalence class.

We use this insight to define a property called *Mediation Dominance*. This property states that solving a cut problem in graph G for level l_1 may solve any overlapping problem in the same graph G for a level l_2 if $l_1 \preceq l_2$. The intuition behind this is that since l_1 is higher integrity than l_2, the semantics of mediation implies that any mediators that can mediate for l_1 can automatically mediate for l_2. Therefore, by solving the cut problems for l_1 before l_2, we get two advantages: (1) we may solve a smaller problem for l_2 (compared to solving the problem for l_2 independent of l_1) since *mediation dominance* enables us to remove the mediators computed for l_1 and any flows they fostered before solving the problem for l_2 and (2) if there is an overlap in the graph between the different cut problems of *comparable* lattice levels, then the size of ordered cut solution can be smaller than the naive solution, a *union* of the individual solutions.

[4]In graph theory, given a graph G=(V,E), an edge cut of this graph with respect to a source and a sink is a set of edges whose removal will divide the graph into two components, one containing the source and the other containing the sink, such that the sink can no longer be reached from the source.

```
1  MediationResolution(G, L, M, MaxRaise) {
2      LS ← TopologicalSort(L)
3      for (l ∈ LS)
4          do { Sources ← {l_i ∈ L | l_i ⋠ l}
5               Mediators ← Mediators ∪
6                   MinimumCut(G, l, Sources, M, MaxRaise)
7               }
```

Figure 4: Greedy Mediation Resolution Algorithm

The algorithm, GREEDY MEDIATION RESOLUTION, that solves a cut problem set of graph G for the lattice $\mathcal{L}$, is shown in Figure 4. The algorithm receives a data flow graph G, a lattice $\mathcal{L}$, the mapping M of nodes to levels, and the *maxraiselevels* for the mapped nodes $MaxRaise$. Line 2 first sorts the levels in the lattice to order the cut problem set based on mediation dominance as described above creating the ordered set of levels LS. The algorithm then chooses the cut problems from the set in order (Line 3), collecting the levels of source nodes that could cause information flow errors for the current cut problem's sink level (Line 4) and computing the minimum cut, given the nodes that map to those levels (from mapping M) and limited by the constraints on possible cuts ($MaxRaise$).

The running time of the algorithm is dominated by the time of computing the MinimumCut for each sink, (for every label in the lattice $\mathcal{L} = \{L, \preceq\}$). The running time of the MinimumCut algorithm is $O(n^3)$, thus, the running time of the MEDIATIONRESOLUTION algorithm is $O(|L|.n^3)$.

5. EVALUATION

In this section, we present the results of a prototype implementation of our approach on the web application system presented in Section 2.1. We first describe the prototype analysis tool and the configuration of the web application system in Section 5.1. We then evaluate the ability of our prototype analysis tool to compute mediation placements for the web application system in Section 5.2. We find that the remote threats of the web application deployment result in the need for 27 additional program entrypoints to be mediated to achieve Clark-Wilson integrity. We also explore the mediation of all possible local threats to the web browser and web server (including the web application), finding that 85 additional mediated entrypoints or 20 IVPs are sufficient to protect these processes.

5.1 Prototype and Experimental Systems

The prototype has an Eclipse front-end that provides access to: (1) parsers to build system-wide data flow graphs using XSM/Flask [9], SELinux [33], and iptables policies;

VMs	Default	Remote	Local
Database Server	281	15	-
Web Server	217	24	56
Privileged VM	335	0	-
Protected Browser	372	15	47
Regular Browser	371	15	-
Total	1576	69	103
Unique Mediators	525	27	85

Table 1: The number of mediators needed for: (1) infrastructure mediation per VM (default); (2) remote threats when application is deployed; and (3) local threats to deployment. For "unique mediators" we count each mediator only once even if it is used in a different VM.

(2) mapping rules to infer integrity level mappings and integrity level constraints for constructing information flow integrity models; (3) the Lemon graph library [34] to compute graph cuts for our information flow model; and (4) a module for generating DIFC-Flume [20] policies with integrity verification and entrypoint mediation from graph cuts. The code breakdown is as follows: (1) 4303 source lines of code (SLOC) for the Bison based parsers; (2) 405 SLOC in Prolog for mapping rules and some parsing; and (3) 4808 SLOC in C/C++ for the last two tasks.

The experimental system includes two hosts, for the web server and web client, which run Xen VM system 4.1. The web server host runs three VMs: a privileged VM (domain 0), a web server VM including an Apache web server and web application, and MySQL database VM. The client host runs two VMs, each configured to run a web browser, corresponding to the protected and regular VMs in Figure 1.

Each VM runs the Linux 2.6.31-23-generic kernel. The Xen hypervisor enforces XSM/Flask policies [9], and Each of the VMs enforce SELinux policies [33]. While all of the OS policies are SELinux, they are independent in the sense that each policy supports a distinct set of applications and these policies do not refer to the interactions among VMs. Also, each VM runs an iptables firewall to govern network communications. We assume that secure communication (e.g., IPsec or SSL) is used to protect any channel that carries application data between hosts. Unprotected channels are given the "External" level (i.e., system-low).

In addition to separating the functionality of different security levels using VMs, we also use features of the SELinux MAC system to protect the web application (on the server) and web browser (on the clients) processes further. For the web application, we use the `mod_selinux` module for Apache to generate separate web application processes to communicate with protected VMs, regular VMs, and other external parties. For clients, we use SELinux policy booleans to set more restrictive permissions. For web browsers, the base permissions (i.e., booleans are off) include access to system files (e.g., `/etc` and `/var`) and the user's home directory (e.g., for plugins).

5.2 Experimental Results

In this section, we show how to use our prototype tool to compute an information flow policy that approximates Clark-Wilson integrity for the experimental system.

Infrastructure Mediation. First, we compute the entrypoint mediation for each of the infrastructure components independently. Recall from Section 3 that the goal is to identify the mediation required for the deployment of any application on the infrastructure. We explore the ability to estimate infrastructure mediation by computing the mediation required of a VM configured to run the target package of the component, such as the Apache web server, MySQL database, or Firefox browser. For example, we configure all VMs with the 12 base modules of SELinux refpolicy 2.20120725, and the servers and web clients are configured with Ubuntu 11.10 server (28 modules) and Ubuntu 11.10 desktop (29 modules), respectively. Upon this, we install the application modules (e.g., four policy modules for Apache are added to the web server VM).

Table 1 (Default) shows our prototype's estimate of the minimal mediation provided by each infrastructure component. For example, the web server VM requires that at least 217 program entrypoints provide mediation to protect the web server process and kernel integrity from remote threats based on the default firewall and MAC policies and given the runtime trace. The estimates for other VMs are somewhat higher, indicating that the web server may be easier to defend than the others.

Note, however, that many of the mediators are common across VMs. There are 525 unique mediator entrypoints across all programs in these VMs (i.e., a program may appear in multiple VMs). In particular, 161 entrypoints are common across all VMs, which is approximately 50-75% of the mediator entrypoints in each of the infrastructure VMs.

One claim is that by examining entrypoint mediation rather than process-level mediation we greatly reduce the number of entrypoints that must be examined. The 122 programs that require mediation above have 2604 active entrypoints[5], meaning that about 20% of all these programs' entrypoints require mediation to prevent illegal information flows in default configurations. Thus, it is beneficial to focus on the individual entrypoints threatened by untrusted input.

Remote Threats to Applications. Table 1 (Remote) shows the entrypoint mediation required to block remote threats when the web application is deployed on this infrastructure. While new mediation is required in most VMs, only 27 new, unique entrypoints need to be protected due to the application deployment. 11 entrypoints are necessary for the new web application programs themselves to receive untrusted input. In addition, 10 new mediators are necessary for auditctl, a program for managing the kernel's audit system. The other six entrypoints occur in the dynamic linker in some programs (setfiles, modprobe, sh, etc.) that now have access to files that may be modified by remote adversaries. As a result, the linker may be prone to new untrusted search path vulnerabilities when the application is deployed. New attack paths for particular application deployments can be identified by this analysis.

The browser is not generally part of the web infrastructure, but we can leverage the idea of infrastructure to simplify mediator placement. First, like all the VMs, the browser has 161 mediators in common with the web infrastructure components. Table 1 (Default) shows the mediators required of our particular browser configurations. While such configurations may differ widely, once the administrator settles on

[5]Note that the number of entrypoints is based on the runtime analysis used to build the connection policies, see Section 4.1. Only the entrypoints actually used in this analysis are included.

a configuration, they can evaluate the default mediation expected. This may be useful should the users want to leverage new software for a particular task. For example, the IceCat browser is a relatively new GNU browser package, and our prototype identifies the entrypoints that require mediation for this new software. We recently found a vulnerability in IceCat because it did not protect an entrypoint [53], so identifying where mediation is required may be helpful in proactively preventing such bugs. Once vetted, the browser configurations themselves may be reused for many deployments, thus becoming part of the infrastructure.

Finally, we note that Xen's domain 0, the privileged VM, has no new mediation required due to the application deployment. This is to be expected as domain 0 is infrastructure for the VMs rather than applications.

Local Threat Mediation. Table 1 (Local) shows the amount of additional mediation sufficient to thwart local threats that may impact the web application. As the privileged and database VMs are deployed from the distribution, we only consider local threats to the web server and the web browser. We identify local threats as those objects (MAC object labels) that may be modified by subjects (MAC subject labels) that are not trusted by the target (web browser or web server). These untrusted subjects are mainly user subjects for running ad hoc programs on these VMs.

Of the 248 possible threats to the web browser and 217 threats to the web server based on the MAC policy[6], only 9 and 11 are unmediated local threats, respectively. That is, the other threats are already blocked completely by mediators placed for infrastructure or remote threats. For these unmediated local threats, we have a choice of mediating using integrity verification procedures (IVPs) or more entrypoint mediation. Recall that a Clark-Wilson IVP takes data as input and produces a certification that the data meets an integrity requirement (i.e., is a Clark-Wilson CDI). Since only TPs can modify high integrity data in Clark-Wilson, a problem in applying IVPs for local threats is that they are modifiable by untrusted subjects. One approach may be to apply IVPs to monitor such objects for unsafe values, at which point mediation is required. Another approach is to remove untrusted writers of such objects from the system. We show the administrator the set of untrusted subjects that can modify these threats, so that the administrator may choose to remove some of the associated subjects' or packages from the component.

For the web server, 56 more mediators are required, whereas for the browser 47 more mediators are required. As can be seen, most of these mediators are unique (85 out of 103), which differs from the mediators selected for infrastructure and remote threat mediation. Thus, local threats appear to present difficult challenges if they cannot be mediated at the source.

Finally, we note that all the objects only writable by TCB subjects or the target subjects may also present local threats if they are installed from untrusted sources. There are nearly 800 object labels for these files in each VM, but we expect that most would be derived initially from the signed package files. Further study is required to ensure that these files are installed from trusted sources, otherwise the threat they pose could be evaluated using the local threat analysis described above.

[6]It is just a coincidence the the number of untrusted object labels and the number of mediators required for the web server infrastructure in Table 1 (Default) is the same.

VM	Nodes	Edges	Parse Policy(s)	Build Graph(s)	Compute Cut(s)
Web Server	2028	6621	1.3	0.19	4.0
Privileged VM	2794	11383	4.3	0.38	6.2
Database Server	2577	10065	3.1	0.32	5.0
Protected Browser	2978	11674	3.8	0.34	5.8
Regular Browser	2978	11700	3.8	0.35	5.7
Remote Mediators	13427	51716	-	-	33.4
Total	13427	51716	16.4	1.58	60.1

Table 2: Performance of computing mediator placement shown per VM for the infrastructure mediation and for system for remote mediation.

Analysis Performance. Table 2 shows that mediation placement performance is practical for the example scenario. For individual VMs, infrastructure mediation can be computed in 5 to 11 seconds per VM. The VMs range in size from 2000-3000 nodes and 6000-12,000 edges. For the web application, the mediation placement can be computed in less than 80 seconds. Breaking the computation down into two steps for infrastructure and remote mediation does not save time, as a mediator placement for the entire information flow problem can be computed in 37 seconds (plus 18 seconds for parsing and graph build as in Table 2.

In terms of scaling the analysis to larger systems, we note two findings. First, there is no difference among the web client systems, so we are able to evaluate information flow integrity using just one such system. Second, we find that the infrastructure mediation required of the privileged VMs is the same as required when the application is deployed. This implies that the privileged VMs may be evaluated separately from the guest VMs. We will explore how to automate identification of such optimizations in future work.

6. RELATED WORK

Deploying web applications to enforce security requirements has been explored mainly by restricting the permissions available to web clients. For example, Tahoma [10] uses server-supplied manifests to describe the permissions authorized for web clients as part of that web application. Alternatively, FlowwolF provides information flow enforcement at each layer in software stack [14]. Also, browsers have been extended with the ability to enforce arbitrary policies, motivated by the OP browser design [13]. However, end-to-end configuration of web applications to satisfy information flow is not yet supported.

Researchers have long known that end-to-end integrity protection can be modeled as an information flow problem. However, classical integrity models [5, 8] rely on formal assurance for all high integrity processes that receive untrusted inputs, but formal assurance methods have not been developed that scale to the size of modern programs. As a result, least privilege [43] has been adopted as the security goal for integrity protection in commodity systems [33, 31, 27, 54]. However, different deployments imply different privileges, so commodity systems try to accommodate this through mechanisms to compute permissions requested by processes [38, 31, 3] and by providing runtime configuration options to specify permissions for options. Nonetheless, it is important that default configurations work in most cases, so researchers have found that the default MAC policies still permit several operations that would violate classical integrity [7]. Meth-

ods to simplify policies through the use of virtualization do not eliminate such information flows [14, 36].

Researchers have recently proposed practical approaches to integrity that approximate classical integrity models [47, 21, 50, 20, 40]. These practical integrity models are all strictly weaker than classical models, in that they do not require formal assurance for code with the authority to protect integrity while receiving untrusted inputs. However, they express restrictions on how such authority may be used by high-integrity programs. For example, the Decentralized Information Flow Control [20, 56] (DIFC) model provides processes with capabilities to make decisions about handling untrusted inputs.

With the emergence of MAC enforcement in commodity operating systems, researchers proposed various policy design tools to help system administrators and OS distributors configure policies [18, 52, 45, 7]. Broadly speaking, these tools enable a policy designer to evaluate compliance using *reachability* to identify whether an adversary can perform an unauthorized operation, even indirectly. Reachability analyses have also been performed for network policies in the form of *attack graphs* [35, 48, 30], but these represent attacker behavior rather than using the system policies. However, defining which operations are unauthorized and resolving any problems are manual tasks. Researchers have recently focused on defending an *attack surface* [16, 23], which is the set of program interfaces accessible to adversaries. The idea would be that if we can minimize the number of interfaces accessible to attackers, we could minimize our defensive effort.

The problem of verifying that a MAC policy satisfies a set of security requirements is a policy compliance problem. In a *policy compliance problem*, a policy is said to *comply* with a goal if all the operations authorized by the policy satisfy the constraints of the goal [25, 11, 24, 15]. The problem is that MAC policies often fail to comply with integrity requirements, as discussed above, so we must repair non-compliant cases. However, any MAC policy changes must also preserve necessary function, and balancing functional and security requirements is computationally complex in general.

7. CONCLUSION

In this paper, we developed a method for computing information flow policies that protect data integrity using mediation for distributed systems that are constructed from multiple, independent components. To protect the system's data integrity proactively, we have developed a mostly-automated method to transform a set of commodity MAC policies into a system-wide policy that provides classical integrity protection, in particular satisfying an approximation of the Clark-Wilson integrity model. To do this, we build an information flow problem and compute mediation by resolving any information flow errors by solving a type of graph-cut problem. We demonstrate our method on a custom web application running on virtualized SELinux hosts, which finds that only 27 additional entrypoint mediators and 20 integrity verification procedures are necessary to provide information flow integrity approximating Clark-Wilson integrity, showing the practicality of computing threats proactively for distributed systems. Solutions can be found in tens of seconds, making the proposed approach practical for many distributed systems.

8. ADDITIONAL AUTHORS

Additional authors: Nigel Edwards (Hewlett Packard Labs, Email: `nigel.edwards@hp.com`)

9. REFERENCES

[1] Alur, R., and Yannakakis, M. Model checking of hierarchical state machines. *ACM Trans. Program. Lang. Syst. 23*, 3 (2001), 273–303.

[2] Anderson, J. P. Computer Security Technology Planning Study, Volume II. Tech. Rep. ESD-TR-73-51, Deputy for Command and Management Systems, HQ Electronics Systems Division (AFSC), October 1972.

[3] `http://fedoraproject.org/wiki/SELinux/audit2allow`, 1996.

[4] Badger, L., Sterne, D. F., Sherman, D. L., Walker, K. M., and Haghighat, S. A. A domain and type enforcement unix prototype. In *Proceedings of the 5th conference on USENIX UNIX Security Symposium - Volume 5* (1995), SSYM'95, pp. 12–12.

[5] Biba, K. J. Integrity Considerations for Secure Computer Systems. Tech. Rep. MTR-3153, MITRE, April 1977.

[6] Chari *et al.*, S. Where do you want to go today? escalating privileges by pathname manipulation. In *NDSS '10* (2010).

[7] Chen, H., Li, N., and Mao, Z. Analyzing and Comparing the Protection Quality of Security Enhanced Operating Systems. In *NDSS* (2009).

[8] Clark, D. D., and Wilson, D. A Comparison of Military and Commercial Security Policies. In *IEEE Symposium on Security and Privacy* (1987).

[9] Coker, G. Xen Security Modules (XSM). `http://www.xen.org/files/xensummit_4/xsm-summit-041707_Coker.pdf`.

[10] Cox, R. S., Gribble, S. D., Levy, H. M., and Hansen, J. G. A Safety-Oriented Platform for Web Applications. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06)* (2006), IEEE Computer Society, pp. 350–364.

[11] Dragoni, N., Massacci, F., Naliuka, K., and Siahaan, I. Security-by-contract: Towards a semantics for digital signatures on mobile code. In *EuroPKI* (2007).

[12] Garg, N., Vazirani, V. V., and Yannakakais, M. Multiway cuts in directed and node weighted graphs. In *in Proc. 21st ICALP, Lecture Notes in Computer Science 820* (1994), Springer-Verlag, pp. 487–498.

[13] Grier, C., Tang, S., and King, S. T. Secure web browsing with the op web browser. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy* (2008), IEEE Computer Society, pp. 402–416.

[14] Hicks, B., Rueda, S., King, D., Moyer, T., Schiffman, J., Sreenivasan, Y., Jaeger, T., and McDaniel, P. An Architecture for Enforcing End-to-End Access Control over Web Applications. In *Proceedings of (SACMAT)* (June 2010).

[15] Hicks, B., Rueda, S., St. Clair, L., Jaeger, T., and McDaniel, P. A logical specification and analysis for SELinux MLS policy. *ACM Transaction on Information and System Security 13*, 3 (2010).

[16] Howard, M., Pincus, J., and Wing, J. Measuring Relative Attack Surfaces. In *Proceedings of Workshop on Advanced Developments in Software and Systems Security* (2003).

[17] Jaeger, T., Butler, K., King, D. H., Hallyn, S., Latten, J., and Zhang, X. Leveraging IPsec for Mandatory Access Control Across Systems. In *Proc. 2nd Intl. Conf. on Security and Privacy in Communication Networks* (Aug. 2006).

[18] Jaeger, T., Sailer, R., and Zhang, X. Analyzing integrity protection in the SELinux example policy. In *USENIX Security Symposium* (Aug. 2003).

[19] King, D., Jha, S., Muthukumaran, D., Jaeger, T., Jha, S., and Seshia, S. A. Automating security mediation placement. In *ESOP* (2010).

[20] Krohn, M. N., Yip, A., Brodsky, M., Cliffer, N., Kaashoek, M. F., Kohler, E., and Morris, R. Information flow control for standard OS abstractions. In *Proceedings of the 21st ACM Symposium on Operating Systems Pr inciples* (Oct. 2007), pp. 321–334.

[21] Li, N., Mao, Z., and Chen, H. Usable Mandatory Integrity Protection For Operating Systems. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy* (May 2007).

[22] Linux KVM. Kernel based virtual machine. `http://www.linux-kvm.org`.

[23] Manadhata, P., Tan, K., Maxion, R., and Wing, J. M. An Approach to Measuring A System's Attack Surface. Tech. Rep. CMU-CS-07-146, School of Computer Science, Carnegie Mellon University, 2007.

[24] Massacci, F., and Siahaan, I. Matching Midlet's security claims with a platform security policy using automata modulo theory. In *In proceedings of NordSec* (2007).

[25] McDaniel, P., and Prakash, A. Methods and limitations of security policy reconciliation. *ACM Trans. Inf. Syst. Secur.* (2006).

[26] Morris, J. New secmark-based network controls for selinux. `http://james-morris.livejournal.com/11010.html`.

[27] MSDN. Mandatory Integrity Control (Windows). `http://msdn.microsoft.com/en-us/library/bb648648%28VS.85%29.aspx`.

[28] Myers, A. C. JFlow: Practical mostly-static information flow control. In *POPL '99* (1999), pp. 228–241.

[29] Myers, A. C., Zheng, L., Zdancewic, S., Chong, S., and Nystrom, N. Jif: Java information flow. `http://www.cs.cornell.edu/jif,July2001-2003`.

[30] Noel, S., Jajodia, S., O'Berry, B., and Jacobs, M. Efficient minimum-cost network hardening via exploit dependency graphs. In *ACSAC* (2003).

[31] Novell. AppArmor Linux Application Security. `http://www.novell.com/linux/security/apparmor/`.

[32] NetLabel - Explicit labeled networking for Linux. `http://www.nsa.gov/selinux`.

[33] Security-enhanced linux. `http://www.nsa.gov/selinux`.

[34] on Combinatorial Optimization, E. R. G. Lemon Graph Library. `http://lemon.cs.elte.hu/trac/lemon`.

[35] Ou, X., Boyer, W. F., and McQueen, M. A. A scalable approach to attack graph generation. In *CCS* (2006).

[36] Payne, B. D., Sailer, R., Caceres, R., Perez, R., and Lee, W. A layered approach to simplified access control in virtualized systems. *ACM SIGOPS Operating Systems Review 41*, 4 (July 2007), 12 – 19.

[37] Pike, L. Post-hoc separation policy analysis with graph algorithms. In *Workshop on Foundations of Computer Security (FCS'09). Affiliated with Logic in Computer Science (LICS)* (August 2009).

[38] Provos, N. Improving host security with system call policies. In *Proceedings of the 2003 USENIX Security Symposium* (August 2003).

[39] Provos, N., Friedl, M., and Honeyman, P. Preventing privilege escalation. In *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12* (Berkeley, CA, USA, 2003), USENIX Association, pp. 16–16.

[40] Roy, I., Porter, D. E., Bond, M. D., McKinley, K. S., and Witchel, E. Laminar: practical fine-grained decentralized information flow control. *SIGPLAN Not. 44*, 6 (June 2009), 63–74.

[41] Rueda, S., Muthukumaran, D., Vijayakumar, H., Jaeger, T., and Chaudhuri, S. Towards system-wide, deployment-specific mac policy generation for proactive integrity mediation. Tech. Rep. NAS-TR-0151-2011, Network and Security Research Center, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA, Sept. 2011.

[42] Sailer, R., Jaeger, T., Valdez, E., Caceres, R., Perez, R., Berger, S., Griffin, J. L., and van Doorn, L. Building a MAC-Based Security Architecture for the Xen Open-Source Hypervisor. In *ACSAC* (Washington, DC, USA, 2005).

[43] Saltzer, J. H., and Schroeder, M. D. The protection of information in computer systems. *Proceedings of the IEEE 63*, 9 (Sep. 1975).

[44] Santos, N., Rodrigues, R., Gummadi, K. P., and Saroiu, S. Policy-sealed data: a new abstraction for building trusted cloud services. In *Proceedings of the 21st USENIX Security Symposium* (2012), USENIX Association.

[45] Sarna-Starosta, B., and Stoller, S. D. Policy analysis for security-enhanced linux. In *WITS* (April 2004).

[46] Schiffman, J., Moyer, T., Jaeger, T., and McDaniel, P. Network-based root of trust for installation. *IEEE Security & Privacy* (Jan/Feb 2011).

[47] Shankar, U., Jaeger, T., and Sailer, R. Toward Automated Information-Flow Integrity Verification for Security-Critical Applications. In *Proceedings of the 2006 ISOC Networked and Distributed Systems Security Symposium* (February 2006).

[48] Sheyner, O., Haines, J., Jha, S., Lippmann, R., and Wing, J. M. Automated generation and analysis of attack graphs. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2002), IEEE Computer Society, p. 273.

[49] Smalley, S., Vance, C., and Salamon, W. Implementing SELinux as a Linux Security Module. Tech. Rep. 01-043, NAI Labs, 2001.

[50] Sun, W., Sekar, R., Poothia, G., and Karandikar, T. Practical Proactive Integrity Preservation: A Basis for Malware Defense. In *Proceedings of the IEEE Symposium on Security and Privacy* (2008), IEEE Computer Society, pp. 248–262.

[51] Sun Microsystems. Trusted solaris operating environment - a technical overview. `http://www.sun.com`.

[52] Tresys. SETools - Policy Analysis Tools for SELinux. Available at http://oss.tresys.com/projects/setools.

[53] Vijayakumar, H., Jakka, G., Rueda, S., Schiffman, J., and Jaeger, T. Integrity walls: Finding attack surfaces from mandatory access control policies. In *Proceedings of the 7th ACM Symposium on Information, Computer, and Communications Security (ASIACCS 2012)* (May 2012).

[54] Watson, R. N. M. TrustedBSD: Adding trusted operating system features to FreeBSD. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference* (2001), pp. 15–28.

[55] Zanin, G., and Mancini, L. V. Towards a formal model for security policies specification and validation in the selinux system. In *SACMAT* (2004).

[56] Zeldovich, N., Boyd-Wickizer, S., Kohler, E., and Mazières, D. Making information flow explicit in HiStar. In *OSDI* (2006).

[57] Zeldovich, N., Boyd-Wickizer, S., and Mazières, D. Securing distributed systems with information flow control. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2008), NSDI'08, USENIX Association, pp. 293–308.

CodeShield: Towards Personalized Application Whitelisting

Christopher Gates
Department of Computer Science and CERIAS
Purdue University
gates2@cs.purdue.com

Ninghui Li
Department of Computer Science and CERIAS
Purdue University
ninghui@cs.purdue.com

Jing Chen
Department of Psychological Sciences and CERIAS
Purdue University
chen548@psych.purdue.edu

Robert Proctor
Department of Psychological Sciences and CERIAS
Purdue University
proctor@psych.purdue.edu

ABSTRACT

Malware has been a major security problem both in organizations and homes for more than a decade. One common feature of most malware attacks is that at a certain point early in the attack, an executable is dropped on the system which, when executed, enables the attacker to achieve their goals and maintain control of the compromised machine. In this paper we propose the concept of Personalized Application Whitelisting (PAW) to block all unsolicited foreign code from executing on a system. We introduce CodeShield, an approach to implement PAW on Windows hosts. CodeShield uses a simple and novel security model, and a new user interaction approach for obtaining security-critical decisions from users. We have implemented CodeShield, demonstrated its security effectiveness, and conducted a user study, having 38 participants run CodeShield on their laptops for 6 weeks. Results from the data demonstrate the usability and promises of our design.

1. INTRODUCTION

Intrusion of end hosts is among the most important computer security problems today. Compromised hosts may be infected with spyware and rootkits. Moreover, compromised hosts are often organized into botnets to carry out attacks such as data collection, phishing, spamming, and distributed denial of service.

In this paper we aim to protect those end host systems that are most in need of protection, namely Window-based hosts used by non-technical users. We observe that almost all existing malware attacks involve downloading malicious code foreign to the host, and then executing the code. If a protection system can stop the *execution* of such foreign code, then these attacks are prevented from succeeding. Antivirus products, probably the most widely deployed security technology, attempt to do this. However, they primarily rely on a blacklisting approach: only programs matching signatures of known malware will be stopped. This blacklisting approach suffers from the limitation that it cannot protect against new malicious code or new variants, and its effectiveness has been shown to be limited [24].

A natural solution is to use a whitelisting approach [21]. Each host should maintain a whitelist of programs, including stand-alone executables and libraries. Only these programs can be executed. Indeed, many commercial whitelisting products exist. Whitelisting is highly effective for blocking malware and unwanted programs from being installed or executed on the system, and it can be applied in settings where systems change very infrequently or in enterprise environments where security is of paramount importance. However, existing whitelisting approaches either use a one-size-fits-all whitelist or require a security expert to configure the whitelisting policy, and thus they cannot be applied to home users or users in organizations where more flexibility is needed. After evaluating the space of whitelisting solutions, Shein concluded [22]: "As yet, no feasible options exist for home users who wish to use application whitelisting."

We propose to use Personalized Application Whitelisting (PAW) to protect end user hosts. In the PAW approach, each system has its own whitelist of programs, and the whitelist is maintained by cues from the end users. The fundamental challenge here is that to serve the diversified need of the user population, end users must be involved in the decision of whether to add a program to the whitelist. At the same time, users lack sophisticated security knowledge and can make mistakes.

To meet this challenge, we examine the pitfalls of existing security mechanisms and identified the following requirements for security interfaces: (1) Reduce the number of times users are asked for a decision. (2) Only ask questions that users know how to answer. (3) Avoid making users passively respond to security prompts. (4) Do not provide an easy and insecure way out.

We satisfy these requirements by introducing the following design features for PAW. (a) Use a whitelist of certificates to automatically accept programs signed by those trusted certificates. (b) Stop the execution of any code not on the whitelist without prompting the user to make a decision. (c) Create a security model in which a user needs to make a high-level security decision, rather than per-program decisions. (d) Force the user to take an active action (instead of responding to a dialog box) for adding programs in the whitelist. The features (a)-(d) together reduce the number of decisions a user needs to make; they also make the safe option the

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

least-resistant path, and the unsafe option requires the user's active attention and a moderate amount of extra work.

More specifically, we propose the CodeShield approach for implementing PAW. CodeShield uses a novel and simple security model. The system is operating in one of two modes: normal or installation. Most of the time, the system is in normal mode, in which any "new" program is prevented from executing. The user can switch the system into installation mode through a trusted path mechanism. In installation mode, new programs can be added to the whitelist. In CodeShield, the security decision a user needs to make is "Do I want to install new software packages on the computer *now*?" This decision is not related to any specific program or resource, and is only related to the user's purpose of using the computer.

The main contributions of this paper are as follows. First, we propose Personalized Application Whitelisting (PAW) as an effective approach protecting systems that are most in need of protection, namely Windows machines used by non-technical users. We introduce the CodeShield approach to making PAW usable by the end users. At the core of our approach is a novel and simple two-mode security model.

Second, we introduce a new way of performing user interactions when end users need to make security-critical decisions. The standard "**warning approach**" prompts the user with a dialog-box when a potentially dangerous action occurs and asks whether the user wants to continue. We propose the "**stopping approach**", which simply stops the potentially dangerous action. The user is informed about the decision, but not asked to intervene in any way. However, the system gives the user the ability to carry out this action through another interface. The advantage of this approach is that in order to carry out a potentially dangerous action, the user must *proactively* carry out the action necessary to do so. This both calls for the user's attention, and adds a level of inconvenience for performing this dangerous action. This user interaction model can be applicable in security features other than PAW.

Third, we have have implemented CodeShield, and have conducted a user study to evaluate it. In the study, 38 participants used CodeShield on their laptops for 6 weeks. The users were divided into two groups. One group used a design which required rebooting to enter installation mode, and the second group used a version of CodeShield that also allows switching directly into installation mode. Our results illustrate the feasibility of the CodeShield approach and that the first group entered installation mode less than the second group. In the literature, we have found few work that develops a new OS security feature and evaluates its usability through a real-world user study. Due to the importance of human factors in security, we think this is an important contribution.

2. MOTIVATION AND RELATED WORK

Blacklisting and Whitelisting. Blacklists are currently the most popular solution to detect and isolate malware. Most of these are commercial products; however there are several research thrusts in this area as well such as CloudAV [19] and AVFS [17] which utilize signatures to detect malicious software. Polymorphic code, packers, and the sheer volume of malware have lead to other problems for signature based solutions. McAfee reports in the second quarter of 2011 [12] as many as 6 million new unique pieces of malware were added to their list, which means about 66,000 a day. In the third quarter this slowed to just around 5 million new unique pieces of malware. BitShred [10] aims to reign in some of these problems, creating more efficient mechanisms to cluster and detect malware as well as identify relevant features of malware to help extract signatures. However the main drawback with any blacklist technology is that new and unknown samples are difficult to detect. This is demonstrated in [24] where detection rates for 6 major commercial A/V products were tested. They visited well known malicious sites to find malware samples, and still only around 60% of malware were detected on the first day. This improves to 85% after 8 days. These numbers were for malwares collected from known malicious sites. For malwares in the wild, this rate is likely to be even lower.

The advantage of application whitelisting has been increasingly recognized [21]. Several products for application whitelisting exist on the market today, such as Windows AppLocker [16], McAfee Application Control [15], and Bit9 [2]. These products, however, primarily target enterprise environments, where security experts configure the policy for hosts. This does not apply to home users or users in organizations that cannot enforce strict one-size-fits-all security policy for all machines (such as a university). One can also view Apple's approach of allowing only Apple approved software on iOS as a whitelisting approach. This, however, uses a one-size-fits-all whitelist, and is not applicable to desktop environments as well as not being conducive to freedom. For example, one challenge is who gets to decide which programs should be on the global whitelist and how are they deciding whether a program is denied or allowed. Our work differs in that we aim at enabling each host to have its own whitelist.

Whitelisting has also been proposed in contexts other than controlling execution of programs. In [7], it is applied to defend against spam emails, where one accepts only emails from whitelisted addresses. When a sender is not on the receiver's whitelist, the sender receives a challenge, which when successfully completed, adds the sender's email address to the whitelist. In [26], whitelisting is applied to defend against web-based phishing attacks.

Disk Protection and Verification. Rootkit Resistant Disks [4] and Bin Locking [29] look at how to protect files that are already on a system from being modified by adding mechanisms to prevent writing unless certain conditions hold. Tripwire [11] uses a static approach to verifying executables by checking a hash of the file to detect modification. DigSig [1] develops a mechanism for Linux to digitally signed binaries and verify the signatures when loading binaries. These approaches, however, do not deal with the end user's need to manage what binaries can be loaded.

Another related work is BLADE [14], which monitors browser disk activity (IE and Firefox) as well as GUI events to differentiate intentional downloads from drive-by downloads. They take cues from user behavior to make a judgment about whether or not the download went through the standard mechanism of asking the user to save the file or not. CodeShield is similar in spirit in that it infers whether a user really intends to run a program; however, instead of controlling only binary programs downloaded by the web browser, we look at the system as a whole and control programs coming from other sources as well.

BinInt [28] works on windows and restricts execution of software except when it was added to the system through a specific channel. To install new programs or update existing ones, the user needs to launch the installer through the command line, via a special privileged command similar in nature to Unix's sudo. Our approach differs in that we use trusted certificates to handle updates and the installation mode for adding new programs, making the user interaction easier for end users.

Human Computer Interaction. Our work benefits from studies on how and why the current mechanisms to communicate security risks to the user are deficient [6, 27, 20]. Motiee et al. [18] investigated the effectiveness of the User Account Control feature in Microsoft Windows. Their results show that 69% of participants did

not apply the UAC approach correctly. User tendencies to dismiss security dialogs have long been recognized, and a number of approaches have been proposed to address these habits. Researchers proposed to disclose threats in plain language and strongly suggest a preferred course of action [30], train users before application use by employing games [23], use polymorphic and audited dialogs [3], reward users' secure behavior instead of penalizing insecure behavior [25]. We propose a different approach to deal with ineffective security dialog: getting rid of the security dialog and changing the way that a user interacts with the security mechanism. Rather than passively responding, the user needs to take active action to install new applications.

3. USER INTERFACE DESIGN OF PAW

In this paper, we propose Personalized Application Whitelisting (PAW), in which each host has its own whitelist for programs, and the whitelist is controlled by the end user. This approach provides the flexibility needed for end user environment, and is more secure than a one-size-fits-all approach in that it exposes each host only to programs on the personalized whitelist.

3.1 User-Centric Design

One key challenge in implementing PAW is that users need to make decisions on adding new programs into the whitelist. However, pushing security decisions to end users has to be done carefully. Given that the vast majority of ordinary users are nontechnical and can easily be confused and/or worn out by repeated security questions, a security defense based upon their discretion appears rather fragile. On the other hand, any system targeting end-users must have the flexibility to accommodate a wide spectrum of different users, and therefore needs to get the human involved in the decision making loop. This dilemma between fallible human nature and inevitable human decision making is the main challenge. Before discussing our design to meet this challenge, we first examine the pitfalls of existing security interface designs, using Microsofts User Account Control (UAC) as the main example. UAC introduces mandatory access control mechanisms with coarse-grained access control decisions to be made by users. The aim of UAC is to improve security by limiting application software to standard user privileges until an administrator authorizes an elevation.

Users are asked too often. In existing systems, users are asked too often for security decisions, and most of the time they need to answer "yes". This makes the users form the habit of automatically answering "yes", often without reading the warning message, let alone carefully considering the decision.

In one study [18], it is shown that 69% of participants did not apply the UAC approach correctly. According to [8] in the first several months after Vista was available for use, people were experiencing a UAC prompt in 50% of their "sessions" - a session is everything that happens from logon to logoff or within 24 hours. With Vista SP1 and over time, this number has been reduced to about 30% of the sessions. The report [8] suggests that "users are responding out of habit due to the large number of prompts rather than focusing on the critical prompts and making confident decisions". However, the data in [8] also suggest that UAC has been effective in forcing application developers to write programs without elevated privileges unless necessary.

Users are asked questions they do not know how to answer and/or presented with information that is difficult to understand. In many security mechanisms, users are told either vague information such as "XX program would like to make changes to your computer", or overly specific information "XX program with

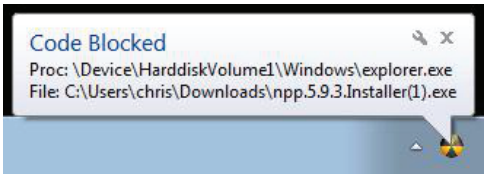


Figure 1: Block notification that is displayed to the user when execution is denied

Y certificate is trying to change your firewall." Most of the prompts presented to the user aren't helping them to make more informed decisions regarding security [9].

Users are made to passively respond to a security question and are provided with an easy and insecure way out. Most security interfaces show a security-warning dialog box to the user, asking the user to passively respond. Furthermore, they allow users to easily follow the insecure path, e.g., press the "continue" button to grant a permission or run a program. One problem of this approach is that the user is "put on the spot" to make a decision. Unless a decision is made, the system is stuck. When forced to make a decision, and presented with confusing and challenging information, it is no wonder users follow the easy and insecure way out.

Based on the above analysis, we learn the following lessons.

- **Reduce the number of times users are asked for decisions.** Users should need to make decision only sparingly.
- **Users should be asked question that they can answer.** The questions posed to the users should be at a level that match users' mental security model.
- **Avoid making users passively respond to security prompts.** When possible, avoid putting the user "on the spot" for a security decision.
- **Do not provide an easy and insecure way out.** When facing a security decision, the less secure option should require more effort to be carried out.

3.2 User Interaction

In our system, the first time a new binary is executed we need to determine whether the user really intended to perform that action. To design a user interaction model that enables the users to make better decisions, we introduce a simple scurity model that has two modes for the system: *normal mode* and *installation mode*. Most of the time, the system is in the normal mode. When the user intends to install new applications on the system, the user switches the system into installation mode, in which newly created programs are added to the whitelist. In our design, the user needs to decide:

> Do I want to install new software *right now*?

This question is not not tied to a particular program, and is posed at a high level regarding the user's usage of the computer.

Next, we need to decide how the user interaction works. More specifically, when the system attempts to load a program that is not on the whitelist, what should happen?

We propose a *stopping approach*. In this approach, when the system attempts to load a program that is not on the whitelist, the loading is blocked, and the user is notified about this event through the notification area on the task bar as shown in Figure 1. If the user really wants to run the program, the user recognizes that a block has occurred and can then switch the system into installation mode, and repeats the action to run the program.

Entering installation mode should be achieved through a trusted path mechanism to prevent malicious code from causing the system to enter installation mode without user consent. We investigate two approaches for entering the installation mode. The preferred approach is to have the system reboot first before it can enter installation mode. In the user study, we also evaluate a secondary mechanism which will enter installation mode without a reboot.

Requiring a reboot has a number of significant advantages. First, when malicious code comes in through vulnerabilities in running processes, our system can stop the future loading of any malicious binary programs that have been written to disk, however the malicious code is still in memory. A reboot clears out any malicious code that might be in memory before the system enters installation mode. Second, when a system reboots, the number of applications that are running is typically smallest, hence the vulnerability surface of the system is also minimized.

Last but not least, introducing new executables into a system is one event that significantly puts the system at risk. A reboot intentionally makes this potentially dangerous event distinct from the normal usage of the system. This hopefully is able to capture the user's attention and highlight the fact that this action introduces significant risk to the host. We believe that the action of entering installation mode must have a certain degree of inconvenience, to serve as a cost that the user has to pay for carrying out the action of adding new software to the system. Allowing a system to easily enter installation mode with little or no inconvenience will cause users to enter installation mode casually, and sometimes unnecessarily. Our user study shows that when given the option of switching into installation mode without rebooting, users enter installation mode about 3 times as often as the users with only the reboot option.

Our user interaction design represents a paradigm shift from the standard approach. Instead of prompting the user to make a decision, the system makes the decision to block the execution. This avoids prompting the users frequently and conditioning them to allow all prompts just so they can continue their flow. Instead, the system allows them to continue their flow without interruption. However, if the user really needs to carry out the action, the user can. Users are not "put on the spot" to make the security decision and can take time to make it. Furthermore, in this design, the more dangerous option of entering installation mode requires an extra step to occur. The "path of least resistance" is the safer option.

4. CODESHIELD

We propose CodeShield, an approach to implement PAW on Windows hosts.

4.1 The Design

We now present our design of CodeShield. A summary of the model, which can be presented to users as a high-level description of the model, is as follows:

> New programs cannot be executed in normal mode. In order to install or execute a new program on the system, enter installation mode.

The detailed workings of the model are summarized as follows.

- The operating system is in one of two modes: *normal* or *installation*. Switching the system from normal mode to installation mode can occur only through a trusted path mechanism.
- When the system is in normal mode, a few specially identified trusted services are marked as *trusted installers*. These typically include the operating system updater.
- The system maintains a list of public keys for trusted software vendors. When a signed program is loaded during installation mode, the signature is added to the list of trusted public keys.
- The system maintains a whitelist of allowed programs. All programs on a system before CodeShield first runs are considered to be on the whitelist. Any program created, updated, or loaded by a trusted installer or during installation mode is added to the whitelist. Any program that is signed by a trusted public key is added to the whitelist.
- A program on the whitelist that is modified in normal mode by a process that is not a trusted installer, is removed from the whitelist.
- In normal mode, when a process that is not a trusted installer attempts to load a program not on the whitelist, the loading is stopped.

4.2 Trusted Certificates and Updates

Our design maintains a list of certificates for trusted vendors and automatically allows programs signed by one of them to be added to the whitelist. Without a mechanism like this, every update must be treated as a new installation since updates will modify executables. We point out that maintaining a list of trusted certificates alone, without the normal-and-installation mode design, is insufficient in today's end-user environments. First, not all programs are signed, and these unsigned programs need to be handled. Second, even when all programs are signed, this only changes the problem of whitelisting programs to that of whitelisting certificates, and the challenges of helping users manage the whitelist remain.

Based on the results of our user study, we have found that most popular applications which are performing updates do sign all their binaries. Perhaps software vendors recognize that the update process needs to be cryptographically protected to prevent malware from exploiting it. Software packages that do not use digital signatures tend to have an update process that is similar to software installation, i.e., downloading a new package and having the user manually execute it. The fact that signing software is becoming more prevalent helps to allow CodeShield to handle updates for software packages once trust has been established in a digital signature. Experiences with signatures will be discussed more in Section 6.

4.3 Implementation

We have implemented CodeShield on Microsoft Windows operating systems. Our implementation works under XP, Vista and Windows 7, and controls the loading of binary programs (including executables and libraries). Our goal with this implementation is to evaluate whether our design is effective in preventing existing attacks and to what degree it is usable.

Our implementation of CodeShield has two parts: a kernel-space filesystem minifilter driver and a user-space client. The minifilter driver is inserted in the stack of the Windows IO Manager, it gets its name from the fact that it can filter IO Request Packets (IRPs) which are used to handle data between the disk and the OS. It intercepts all communications to the disk and takes action for certain events. This is the same mechanism that disk encryption and antivirus applications employ.

Blocking Execution CodeShield needs to get into the decision path of when to allow code to load. Microsoft's recommendation for independent software vendors to block the loading of code modules, is that "Intercepting IRP_ACQUIRE_FOR_SECTION_SYNCHRONIZATION and

returning STATUS_ACCESS_DENIED when sections are loaded for PAGE_EXECUTE permission is an appropriate approach." [5] The IRP to synchronize a section from disk into memory with the execute flag set indicates that a file is being loaded for execution. On versions of Windows with Data Execution Prevention (DEP) turned on, this is a requirement for execution. CodeShield applies the whitelisting policy when intercepting these events, and prevents the synchronization from disk to memory if the file is not on the whitelist.

Detecting modifications and maintaining the whitelist The minifilter intercepts file system events by catching a set of IRPs that indicate that a write has occurred. Our implementation uses the alternate data streams associated with files, which are part of NTFS, to identify whether the files are on the whitelist or not. In this way the whitelist information is attached to a file and follows the file even if it moves. We can choose to mark each file that is on the whitelist, or each file that is not. We chose the latter in our implementation to avoid having to scan the hard drive to mark every file when CodeShield first installs. For each file that has been written, CodeShield writes in the alternate data stream what processes have contaminated that file. Any such contaminated files are considered not on the whitelist. When a file is accessed in installation mode, by a trusted installer, or if the file is signed by a trusted certificate, this contamination is removed. When the kernel is not compromised, all requests to modify anything on disk pass through the CodeShield minifilter which can deny access to the alternate data streams utilized by our system.

While CodeShield marks files that are not on the whitelist, it is maintaining a logical whitelist. CodeShield marks a file not because the file is on some pre-defined blacklist, but rather because the file has been written to (i.e., either newly created or modified), and hence should not be on the whitelist. Any file without contamination is either already on the system before CodeShield is installed and has not been changed, or has been added to the whitelist by CodeShield.

CodeShield currently only supports a system where we can write to meta-data for files, this is only feasible for the local disk. Since our target is normal end-users and a proof of concept, this is a reasonable constraint. However for networked drives, CDs, and other types of media, this may not be reasonable. One possibility is to have CodeShield maintain an extended whitelist with hashes of whitelisted files that are not local to the system or on drives that don't support our meta-data.

User Interaction CodeShield needs to interact with users. There is a dedicated user space client which displays dialog boxes, task bar notifications, and sends responses back to the kernel mode minifilter driver. The client displays a notification from the task bar whenever a load is blocked as seen in Figure 1. Through the client, the user can initiate a reboot/switch into installation mode, examine the list of trusted certificates, the list of events monitored by CodeShield including blocked executions, files added to the whitelist, and writes to certain files (i.e., removal from the whitelist). Due to the availability of APIs, we chose to handle the processing and verification of all digital signatures in the user space client, which reports information back to the minifilter driver.

Installation Mode. Although our design calls for a Trusted Path mechanism to enter installation mode, we cannot tap into Windows' Secure Desktop to do it. In our current implementation, the "reboot into installation mode" mechanism utilizes a registry key that is managed by the minifilter driver to pass the intent to enter installation mode through the reboot. This registry key is protected in the same way that the alternate data streams are, by denying access to write to the registry key, this time using a RegistryCallback routine from the driver. In our user study, we also evaluate a version that allows a user to switch directly into installation mode.

Whichever way a system enters installation mode, a user specifies a duration for how long they expected to be installing software. When this time has expired, the system attempts to exit installation mode, however, if the user is still installing they have the opportunity to stay in installation mode by specifying how long they would like to remain in installation mode. If they fail to respond to the dialogue, then after a preset amount of time the system simply exits installation mode. The amount of time ranges from 5 minutes to 30 minutes. A user gains little by always responding to stay in installation mode because they will be bothered again soon to exit, and typically the next needed installation is likely days if not weeks away.

5. SECURITY EVALUATION

In this section, we evaluate the security protection provided by CodeShield. In Section 5.1 we discuss the attacks that CodeShield can defend against and look at the current ways in which malware enters a system. We evaluate CodeShield against real attacks in Section 5.2. Finally, in Section 5.3 we look at how CodeShield could be circumvented, in doing so we give the threat model we are assuming in order for CodeShield to offer effective protection.

5.1 Attacks that CodeShield Stops

The security advantage of a whitelisting based defense against malwares has been broadly recognized; however, so far whitelisting has been limited to enterprise environments. Our main contribution is a design to enable personalized whitelisting.

A PAW system such as CodeShield improves the defenses against many of today's most common attacks. As CodeShield blocks all unintended execution of foreign binary programs, it stop drive-by download attacks. More importantly, CodeShield provides system-wide, rather than browser-specific protection for this type of attack. No matter how a malicious program is dropped on to the system, whether it comes in as an email attachment, via exploiting a buggy document reader to drop files, or exploiting some vulnerable services, CodeShield stops the execution of the malicious program.

CodeShield also significantly raises the bar on social engineering attacks. Under CodeShield, installation of a new application is an action distinct from others. If the social engineering attack attempts to exploit the user's confusion to install it, rather than convincing the user to intentionally install it, such an attack will fail. Furthermore, CodeShield's design of using a reboot to enter installation mode has the potential of protecting users from themselves. In situations where that a user impulsively, though still intentionally, downloads an apparently interesting program and wants to run it, CodeShield forces the user to slow down, break the flow of the current activity, go through the highly distinctive and slightly cumbersome steps of entering installation mode, reflecting the danger of such action. Hopefully, this will cause the user to reconsider the impulsive decision to run the program and continue only if installing the new software is really needed.

We also conducted a more systematic analysis to identify to what degree attacks involve new malicious executables. For this, we use McAfee's Threat Intelligence website[1] to determine attack behavior of different types of malware. We manually examined all malware entries on McAfee's Threat Intelligence website over a period of a month. There were approximately 44,000 entries, where each

[1]http://www.mcafee.com/us/mcafee-labs/threat-intelligence.aspx

Total	Write Executable	File is Executable	Unknown
1981	1854 (93.6%)	105 (5.3%)	22 (1.1%)

Table 1: Data Collected from McAfee's Threat Intelligence Website

entry is a different file. These entries are grouped by their detection names, which clusters files that exhibit the same behavior or are part of a single attack. The were 1981 total detection names. For each detection name, we analyze the description of how each attack modifies the system as reported by McAfee, the results are presented in Table 1. In the first type, threats write some executable files to disk, e.g., a dll, exe or sys file, and this accounts for 93.6% of attacks. We expect CodeShield to prevent these attacks. In the second type, the malware sample itself was an executable, but it did not write additional executables to disk. These attacks typically make changes to the registry to cause themselves to be automatically loaded when the system reboots. This type of threat was identified either directly through McAfee's website or by looking up the file hash in VirusTotal[2]. This accounts for 5.3% of the detected attacks. In 1.1% of the cases we could not come to a conclusion based solely on the information collected.

Overall this points to the fact that nearly all malware samples (98.9% in the dataset we analyze) are using executables at some point in their life cycle, either in the initial attack, or to maintain and expand their foothold in the system. So preventing the execution of foreign code would diminish an attackers ability to achieve their goals.

We also stress that a key strength of CodeShield is its ability to protect against new malware that have not yet been detected. This is difficult for traditional A/V products to achieve with their signature based detection mechanism.

5.2 Evaluating the Implementation

While we expect CodeShield to block execution of malware, we also experimentally evaluated CodeShield to verify that this is the case. Through actual usage of CodeShield, we know that CodeShield blocks the execution of downloaded and modified binary programs. We also experimentally evaluated how CodeShield behaves against actual malware samples.

For this evaluation, we use a collection of malware samples collected from Contagio Malware Dump. Contagio collects malware samples from a variety of sources as they occur, and helps the research community understand, analyze and collaborate on those threats. The samples that they collect represent many of the major attacks that have occurred in the past few years including Operation Aurora, Conficker, Zeus and many others.

We tested these malware samples in a VM running Windows XP and using whatever software appropriate to allow the exploit. Microsoft Office 2003 with certain versions of flash, and Adobe Reader 8 and 9 are generally sufficient to allow most exploits to succeed; newer versions of these products are also vulnerable to most of the exploits we tested. The malware samples are in the form of pdf, xls, doc, and ppt files. These samples are used by targeted social engineering attacks, with a filename that would entice the user to open the file. Once the document is opened, these documents drop their files to disk and attempt to execute them. We see in the log files and CodeShield notifications that files are written, and then followed immediately by an execution block.

[2]http://www.virustotal.com/

Date Found	CVE	Ext	Files Written	Stop
2010/03/31	2006-0020	ppt	temp\svochost.exe	YES
2010/02/18	2006-2389	doc	temp\lsass.exe	YES
2010/02/20	2006-2492	doc	temp\svohost.exe	YES
2010/02/22	2006-6456	doc	temp\taskmgr.exe	YES
2010/03/24	2007-5659	pdf	system\32wumsvc.dll	YES
2008/05/12	2008-0081	xls	temp\XLS.exe	YES
2010/01/12	2008-0655	pdf	Temp\1.exe	YES
2010/03/21	2008-2992	pdf	Temp\ews.exe	YES
2010/04/23	2008-4841	doc	temp\svchost.exe	YES
2009/10/02	2009-0658	pdf	system32\ authomal.exe	YES
2009/06/11	2009-0927	pdf	funparams.exe	YES
2010/04/02	2009-0927 2007-5659	pdf	temp\svchosts.exe	YES
2010/05/06	2009-1129	ppt	system32\dxmfnt.exe system32\dxmgnt.exe	YES
2009/10/15	2009-1862	pdf	system32\ authomal.exe	YES
2010/05/10	2009-3129	xls	temp\wuauclt.exe	YES
2010/01/04	2009-4324	pdf	temp\1.exe	YES
2010/11/29	2009-4324 2009-0927 2008-2992	pdf	c:\a.exe	YES
2010/04/28	2010-0188	pdf	Microsfot\ AdobeARM.exe	YES
2010/11/18	2010-1297	pdf	java.exe	YES
2010/10/08	2010-2883	pdf	Temp\AcroRd32.exe	YES
2011/01/07	2010-3654	pdf	windows\update.exe	YES

Table 2: Live Malware Exploits/Samples Tested on a system with CodeShield. CodeShield blocked all of the samples we tested. For space, we only specify the last parts of the paths.

Table 2 shows an analysis of 22 different exploits and a corresponding attack. We limit our reporting to a single instance per exploit, but all instances of an exploit that we tested were blocked successfully. We tested 175 pdf samples that utilized various exploits, we were able to trigger the attack in 141 (80%) of those samples. Of those 141 samples, we blocked 100%. The reason for our inability to trigger all the attacks could be due to several things, including version differences between the exploit and our software, patches applied to our system, or lack of network connectivity on our test machine. Overall this shows the effectiveness of CodeShield mechanism in blocking real world attacks on end user systems.

5.3 Possibilities of Evading CodeShield

If the approach CodeShield uses is widely deployed, it would motivate the attackers to find ways to evade it. We now discuss such possibilities.

Given that CodeShield adds new code that is executed during the installation mode on the whitelist, an attacker may attempt to compromise an existing process, store payloads on disk and wait until the system is in the installation mode to execute. For example, the malicious code may keep trying to load the stored program until it succeeds. CodeShield's reboot-to-enter-installation design can stop this attack. The reboot mechanism delays entering into installation mode until the system is up, the client is connected, and the user responds. So anything that tried to load early in the boot process will be blocked if it is not on the whitelist. To succeed, the malware has to change the start configuration files in a way that loads a malware program with a certain delay after system reboot. If such configuration changes are possible, then CodeShield needs to be enhanced to prevent auto-loading of programs when in installation mode.

An attacker may try to compromise the private key of some trusted vendors and sign the malware. Indeed, evidence has shown that some vendors with legitimate certificates can be malicious, and certificates owned by non-malicious vendors can be compromised [13]. While all security mechanisms that use public keys suffer from this threat, we point out that CodeShield's design makes it less vulnerable than many other mechanisms. For example, Microsoft Windows and Android use a public key infrastructure in which a software vendor can get a certificate from one of the Certificate Authorities (CAs), often with a small payment, and then sign its own programs. Every machine trusts any software signed by any vendor with a valid certificate. In essence, all machines use the same global whitelist of certificates, and the list is often very big. In CodeShield, each host trusts only the certificates from which one has already installed softwares, instead of all vendor certificates. Hence the impact of such attacks is limited unless the attacker is able to compromise the private key of a popular vendor (such as Microsoft or Adobe), which is a serious threat whether CodeShield is there or not.

We note that any attack that does not need to write files to disk to achieve its attack objective is not prevented by CodeShield, as CodeShield is designed to prevent a persistent compromise of a system by preventing unintended binary execution. Therefore, if an attack first exploits a vulnerability in the system and gains admin privilege without loading no binary, the attack can try to disable CodeShield.

Currently CodeShield blocks the execution of binary programs. We focus on binary programs because today's malware programs come in binary form. The attacker can try to deliver malware payloads using intermediate code or script code such as Jar files, .NET CIL files, Python scripts, etc. This is a limitation for the current implementation of CodeShield. We point out that the same limitation applies to all application whitelisting approaches. Also, the concept of preventing the loading of external code can be applied to other forms of code. One could extend the implementation to identify the runtime environment for these types of intermediate files, and control their loading events.

6. EXPERIENCES USING CODESHIELD

As of June 2012 CodeShield has been used on 65 machines, many for 6 weeks, and others for several months. The longest usage is on the laptop of an author of the paper, which has been running different versions of CodeShield for about 11 months, with the most recent version being used continuous for 203 days until the time of writing this paper. Other users include friends/family of the authors, other researchers, and 38 college students who participated in a 6-week user study. The user study was completed in November 2011. Among the 38 participants, 7 have been continuously using CodeShield until the time of writing this paper, which is 3 months after the study has completed. Among the 7 participants, 5 were using a version of CodeShield where rebooting is the only way to enter installation mode. From these experiences, we learned that the key to the success of CodeShield is the ability to handle software updates smoothly. If programs from software updates can be automatically added to the whitelist, without the need to use installation mode, then the inconvenience caused by CodeShield is minimal. In fact, for the two longest users, one entered the installation mode 8 times in 203 days, and another entered the installation mode 13 times in 168 days. In computers used by family/friends of the authors, we often observe the pattern of going for weeks without the need to enter the installation mode.

Many programs that frequently update themselves digitally sign all new binaries, resulting in a smooth experience for CodeShield users. These include software from vendors such as Google, Mozilla, Adobe, and so on. We also note that the behavior of Firefox update changed at some point between March 2011 and the summer of 2011. In March 2011, Firefox auto-updates resulted in unsigned DLLs that were blocked by CodeShield. At first, denying the loading of these DLLs did not cause noticeable differences. (In fact, a somewhat unexpected lesson we learned is that many applications can survive the block of some DLLs.) However, once Firefox autoupdated to Firefox 4 in March 2011, failure to load these unsigned DLLs caused Firefox to have strange appearances, and we had to enter installation mode and run Firefox to add those DLLs to the whitelist. Since Firefox 4 in March 2011, however, all binary files created by Firefox auto-updater are signed. Firefox has been updated to Firefox 10, and CodeShield handled all subsequent updates seamlessly. This is one anecdotal evidence that frequently-updated softwares are increasingly signing the updated files.

We now discuss the experiences we have had when updates may introduce unsigned binaries.

First, Microsoft's automatic updates may create unsigned files. From a security researcher at Microsoft Research, we learned that Microsoft has an internal mechanism to verify the integrity of such unsigned binary files, the interface is currently not available to other developers, but may become available in the future. CodeShield currently does not have access to such an interface. To avoid blocking these Microsoft programs, we currently add three exception rules for *wuauclt.exe*, Windows auto update checker, *trustedinstaller.exe*, Windows trusted installer, and *drvinst.exe*, which handles plug and play driver installations.

Second, there are several anti-virus vendors that do not sign all the code they distribute. It's likely that they are also using their proprietary mechanism to verify the integrity of these files; however, CodeShield is unable to tell and will block these files. Entering installation mode on a daily basis to allow the update to occur would cause an unacceptable overhead. The solution we chose during the user study was to add additional trusted installers to CodeShield for these A/V products. This allowed the A/V software to update smoothly, and not force the users to frequently enter installation mode. This affects our user study results. Before the study, we emailed the participants to provide to us their A/V product name, and test whether their update result in unsigned binary files, in which case we need to add exceptions for them. Some users, however, did not respond to our query and did not contact us during the study; thereby having to enter installation mode frequently to deal with A/V updates.

The final unexpected issue was related to gaming, which we discovered only after receiving the study results. There are game services such as Steam and EA Origin that update frequently with unsigned files. This caused the users who played certain games to enter installation mode much more frequently. This skewed the data in both groups as it forced them to enter installation mode before playing the game, which some users do almost every day. However this overhead didn't seem to effect their gaming, rather just caused them to enter installation mode more often.

We point out that if CodeShield or something similar is integrated into the OS, or a major A/V product, this can start an ecology change towards motivating software vendors to digitally sign all their binary programs, just like the introduction of UAC motivated developers to request elevated privileges less often. Signing all code that one distributes is feasible, as this is already done by many vendors. Also, this is a requirement for many mobile platforms already, and the next generation of Windows is heavily tilted towards mobile platforms.

7. USABILITY EVALUATION

We have conducted a user study to evaluate the usability of CodeShield. In the study, participants were asked to run CodeShield on their primary laptop for 6 weeks. We recruited 38 participants by posting multiple flyers on campus and offering cash payment for participation. The participants came from the student population, and due to the fact that the flyers were spread out in buildings that housed different disciplines, the participants' background were varied. They include not just engineers and scientists, but many liberal arts students, hospitality management students, and other non technical fields.

We divided the participants into two groups. There were 20 participants in the *Reboot Only Group* (reboot group); they were given the design which made the user reboot to enter installation mode. There were 18 participants in the *Reboot or Switch Group* (switch group); they were given a version of CodeShield that had two mechanisms to enter installation mode; reboot into installation mode or switch directly into installation mode. We compare data from the two groups to evaluate the design of requiring a reboot to enter installation mode.

There were 13 training sessions across four days, each session lasting about 30 minutes. Participants were assigned to sessions based on when they entered the study. Participants in each session were assigned to the same group, and we alternate the sessions between group 1 and 2 to balance the number of users in each group. During the training session CodeShield was explained and demonstrated, performing tasks such as installing new software, updating existing software, and what to watch out for if an attack did occur. The participants then installed CodeShield on their laptops, took a brief survey, and began to use their system which was now running CodeShield. After 6 weeks a final survey was automatically presented and we collected their logs and survey results via online submission through the client.

The initial survey collected basic demographic information from the users. All but one user was between the ages of 18-25, the other was 25-35. We asked the users to describe their technical level, 26 stated that they are "proficient with computers", 6 use their computer "just for email and internet, not much else", and 3 considered themselves to be a "programmer or expert". There were no security experts. Laptops are for 'personal' and 'school' use, and about half of these machines are also used for 'work'. Regarding the age of machines, 19 laptops were more than 1 year old, 7 were 6 months-12 months, 3 were 2 months-6 months, and 6 were less then 2 months. All but 6 of these machines were used by a single user. 4 users admit to ignore/disabling update prompts.

The CodeShield program logged all code that was blocked, number of times that a user entered installation mode and how they entered (reboot or direct switch), all major code that was executed (just exes, not dlls), all critical writes (writes to a dll/exe/sys file), system start and shutdown events, and what digital signatures were trusted by CodeShield, and therefore by their system.

Two participants' log files were deleted for unknown reasons and were therefore useless. One other participant never submitted their results. All three belong to the switch group. Hence we are reporting 35 users' data, with 20 users in the reboot group, and 15 users in the switch group.

Among the 15 users in the switch group, two had log data for about four weeks, rather than six weeks. One participant terminated the study after about one month. He reported that his laptop did not boot, and thought it was due to CodeShield. He then reported that the problem went away after reseating the memory chips in his laptop; but after a few days, he decided to withdraw from the study. He nonetheless submitted his log and completed the final survey. Another user dropped his laptop about one month into the study. However, he submitted his log periodically, hence we have about 1 month of his data; he later took a separate final survey, enabling us to include his data. While it is unlikely that all five problem users come from the switch group, by chance, this is exactly what happened.

7.1 User Study Results

While we were confident about the design philosophies of CodeShield, and tested extensively before the user study, we were somewhat concerned about how the actual user experiences would turn out during the 6-week study. As discussed in Section 6, some software updates introduce unsigned files that are blocked. CodeShield is an intrusive security mechanism in that it can prevent any binary from loading and changes how the user interacts with the system. If unexpected events occur in the user's behavior or in software he or she is using, then the system could become quite difficult to interact with. So the possibility of running a busy help desk for six weeks or dealing with potentially disastrous system freezes existed, as it turned out however the user study ran smoothly.

Table 3 presents the summary results for two groups. First we note that the number of times users enter installation mode varied, but users in the switch group enter installation mode much more frequently due to the ease with which they could switch. The max for the switch group was 41, caused by a game that frequently writes unsigned binaries. There were several other participants in the switch group who enter installation mode many times (e.g., 33, 28, 27). The max for the reboot group was 30; this participant failed to inform us about their A/V product and had to enter installation mode to deal with updates. The next highest in the reboot group is 11.

The general response to CodeShield was positive. On a scale of 1-10, where 1 is Strongly Disagree, 5 is Neutral and 10 is Strongly Agree, participants in both groups rated their system as usable. The mean of the reboot group was 7.26, and switch group was 8. Participants were generally indifferent about whether or not they would like to continue using CodeShield, with responses close to Neutral. And we have already mentioned, 7 of the 35 users (20%) continue to use CodeShield at least 3 months after the study has completed.

One of the goals of CodeShield was to create a security mechanism that was closer to the users' mental model. The reason is that if users understand what they are doing, and can map that to a simple security decision, then they are less likely to make dangerous mistakes. Although not fully answered by our study, we believe that the answer to the question "I understand the design rationale of CodeShield", group averages around 8-8.5, shows that there is a general understanding and acceptance for the model presented by CodeShield.

For all the survey questions the responses were generally positive, with the responses from the reboot group slightly less positive then the switch group across all questions. We believe this is a reflection of the overhead in entering installation mode for the reboot group, but due to the size of the study there are no real statistically significant results to report.

One of the most interesting observations is that the reboot group entered the installation mode significantly fewer number of times than the switch group. The average number of times the reboot only participants entered installation mode was 5.25, whereas the average among the switch group was 16.66. The difference in the median, which is less affected by one or two outliers, is even more pronounced: 3.5 for the reboot group and 17 for the switch group. We also observe that in both groups, about 20% of installation modes

	Reboot Group			Switch Group		
Survey Question Range 1-10	**Median**	**Mean**	**StDev**	**Median**	**Mean**	**StDev**
Q1: The system was usable with CodeShield installed.	7.5	7.25	2.35	9	8	1.85
Q2: I would continue to use CodeShield after the study if that was an option	4.5	4.8	3.01	5	5.2	2.95
Q3: I felt like the extra work to get into installation mode was worthwhile	5	4.7	2.61	7	6.46	2.29
Q4: I understand the design rationale of CodeShield	9	8.65	1.46	8	8.06	1.90
Q5: I was surprised about the things that were blocked.	7	6.35	2.70	5	5.8	3.29
Q6: I would like Windows to have the features provided by CodeShield	5.5	5.85	2.25	8	7.2	2.30
Q7: I want to control what new programs can be run on my system	8	7.85	1.46	9	8.67	1.44
Q8: I would recommend CodeShield to friends and family	5.5	5.7	2.49	6	6.27	1.94
Use Statistics	**Median**	**Mean**	**StDev**	**Median**	**Mean**	**StDev**
S1: Average Days Between Installation	12.15	17.68	13.48	3.24	9.98	13.55
S2: Total Installation Mode Switches	3.5	5.25	6.48	17	16.66	12.10
S3: Useful Installation Mode Switches	3.5	4.25	4.21	13	13.13	11
S4: Unique Digital Signatures on System	72	73.15	19.15	84	82.53	26.26
S5: Total Reboots	36.5	44.05	25.14	62	64.8	51.23
File Statistics	**Median**	**Mean**	**StDev**	**Median**	**Mean**	**StDev**
S6: Unique Files WhiteListed due to Signatures	345.5	394	186	509	509	333
S7: Unique Files WhiteListed due to Trusted Installer	232.5	207	94	230	218	115
S8: Unique Files WhiteListed due to Installation Mode	183	248	352	827	769	690

Table 3: This table breaks down several data points about behavior of users in the Reboot Group and the Switch Group. The first part reveals final survey questions and statistics for the responses. Next are statistics regarding frequency of entering installation mode and number of digital signatures on each machine. And finally there are general statistics about how often code entered the whitelist and for what reason.

see no programs added to the whitelist; hence they are unnecessary.

We also note that although the mean switches to installation mode for each group are far apart, the gap between the degree of acceptance of CodeShield is smaller. On all questions reflecting satisfaction with CodeShield, the average of the reboot group were close to but slightly lower than those of the switch group.

Our interpretation for these results is that it is likely that most of the entries into installation mode for the switch group are unnecessary, meaning that the system will continue to function fine without them. In fact, most users can use their computer with very sparse usage of the installation mode. However, because currently CodeShield shows notifications for blocking or every dll, when a user sees that something is blocked while doing a legitimate action, the user enters installation mode to allow it. When directly switching to the installation mode is an option, the user is more likely to enter installation mode. For the reboot group, the user is less likely to actually enter the installation mode due the inconvenience; however, this may cause the user to somewhat dislike CodeShield.

We believe that these results confirmed our conjecture that a level of inconvenience for entering the installation mode is both necessary and beneficial. However, we should refine CodeShield to avoid making users feel that they need to enter the installation mode. For example, we should avoid repeated notifications when multiple DLLs are blocked for the same process.

The mean number of trusted certificates on each machine was 77, indicating many software vendors are signing their code. Even more interesting is the fact that 798 unique certificates existed on all 35 machines participating in the study. 345 appeared on 2 or more machines. That leaves 453 certificates that appeared on a single machine. This highlights the diversity of software installed. This is also the reason that we believe a Personalized Application Whitelisting approach is advantageous to a global approach.

Our data also included how many times each machine reboots during the study, and this reboot count is dominated by those unrelated to CodeShield. In the reboot group the average was around 44 times in the 42 day study, and the switch group surprisingly had more reboots, average of 65 (and a higher standard deviation), even though this group wasn't being made to reboot by CodeShield. We found these numbers unexpectedly high, as our perception is that most people merely close the laptop lid for the night. This finding means that the attack of injecting malicious code only in memory has only a short-term control over a system. This also suggests that the few additional reboots required by CodeShield could be acceptable.

8. CONCLUSION AND FUTURE WORK

We have demonstrated CodeShield as a practical system for non-technical users to protect themselves from a variety of threats. CodeShield detects and mark new data as it is added to disk, and blocks any new content from being executed unless it has been added to the systems personalized application whitelist. The system also provides the user with a more intuitive model to understand what they are currently doing on their system, namely instead of fine grained decisions about specific applications they can merely decide "whether to install new software now." If they decide that is what they are trying to do, then they will put forth the effort to install new software. If they decide that they are not installing new software then they do not have to do anything and the execution will be stopped, keeping the system in its current state. The results of our user study suggest that these installation events are infrequent for most users, so even though installation now incurs an additional effort, that this is justified in order to achieve a more secure system.

The types of attacks that CodeShield will prevent are of critical importance to both average home users as well organizations. Anything that tries to add an executable in a stealthy way, such as drive-by downloads, malicious documents or compromised external media, will be blocked since the execution of the new binary will be stopped. Social engineering attacks which exploit the user will be harder as well because the perceived inconvenience should outweigh the potential benefit, but also because techniques which try to confuse the user will also be more likely to fail because our default is to deny. CodeShield has been tested on several different types of attacks and all were stopped.

9. ACKNOWLEDGMENTS

This work was supported in part by Army Research Office under the grant "Towards a Scientific Basis for User Centric Security Design". Work by C. Gates and N. Li were also supported by the National Science Foundation under Grant No. 0905442.

References

[1] A. Apvrille, D. Gordon, S. Hallyn, M. Pourzandi, and V. Roy. Digsig: Runtime authentication of binaries at kernel level. In *Proceedings of the 18th USENIX conference on System administration*, pages 59–66, Berkeley, CA, USA, 2004. USENIX Association.

[2] BIT9. Bit9 parity suite: Adaptive application whitelisting. `http://www.bit9.com/products/bit9-parity-suite.php`.

[3] J. C. Brustoloni and R. Villamarín-Salomón. Improving security decisions with polymorphic and audited dialogs. In *SOUPS*, pages 76–85, 2007.

[4] K. R. Butler, S. McLaughlin, and P. D. McDaniel. Rootkit-resistant disks. In *CCS '08*, pages 403–416, New York, NY, USA, 2008. ACM.

[5] M. Corporation. Kernel data and filtering support for vista sp1 / windows server 2008. *MSDN*.

[6] L. F. Cranor. A framework for reasoning about the human in the loop. In *Proceedings of the 1st Conference on Usability, Psychology, and Security*, pages 1:1–1:15, Berkeley, CA, USA, 2008. USENIX Association.

[7] D. Erickson, M. Casado, and N. McKeown. The effectiveness of whitelisting: a user-study. In *Conference on Email and Anti-Spam*, 2008.

[8] B. Fathi. Engineering windows 7, October 2008. MSDN blog on User Account Control.

[9] S. Furnell, A. Jusoh, and D. Katsabas. The challenges of understanding and using security: A survey of end-users. *Computers & Security*, 25(1):27–35, 2006.

[10] J. Jang, D. Brumley, and S. Venkataraman. Bitshred: feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 309–320, New York, NY, USA, 2011. ACM.

[11] G. H. Kim and E. H. Spafford. The design and implementation of tripwire: a file system integrity checker. In *Proceedings of the 2nd ACM Conference on Computer and communications security*, CCS '94, pages 18–29, New York, NY, USA, 1994. ACM.

[12] M. Labs. Mcafee threats report: Third quarter 2011, Nov. 2011. White paper from McAfee.

[13] N. Leavitt. Internet security under attack: The undermining of digital certificates. *Computer*, 44:17–20, 2011.

[14] L. Lu, V. Yegneswaran, P. Porras, and W. Lee. Blade: an attack-agnostic approach for preventing drive-by malware infections. In *ACMCCS '10*, pages 440–450, New York, NY, USA, 2010. ACM.

[15] McAfee. Application control. `http://www.mcafee.com/us/products/application-control.aspx`.

[16] Microsoft. Applocker. `http://technet.microsoft.com/en-us/library/dd548340`.

[17] Y. Miretskiy, A. Das, C. P. Wright, and E. Zadok. AVFS: An on-access anti-virus file system. In *Proceedings of the 13th USENIX Security Symposium (Security 2004)*, pages 73–88, San Diego, CA, August 2004. USENIX Association.

[18] S. Motiee, K. Hawkey, and K. Beznosov. Do windows users follow the principle of least privilege?: investigating user account control practices. In *SOUPS*, 2010.

[19] J. Oberheide, E. Cooke, and F. Jahanian. CloudAV: N-Version Antivirus in the Network Cloud. In *Proceedings of the 17th USENIX Security Symposium*, San Jose, CA, July 2008.

[20] F. Raja, K. Hawkey, P. Jaferian, K. Beznosov, and K. S. Booth. It's too complicated, so I turned it off!: expectations, perceptions, and misconceptions of personal firewalls. In *SafeConfig '10*, pages 53–62, New York, NY, USA, 2010. ACM.

[21] R. Shein. *Chapter 1: Whitelisting for Endpoint Defense*, pages 3–14. Auerbach Publications, 2011.

[22] R. Shein, H. F. Tipton, and M. Krause. *Information Security Management Handbook, Sixth Edition, Volume 5*. Auerbach Publications, 2011.

[23] S. Sheng, B. Magnien, P. Kumaraguru, A. Acquisti, L. F. Cranor, J. I. Hong, and E. Nunge. Anti-phishing phil: the design and evaluation of a game that teaches people not to fall for phish. In *SOUPS*, pages 88–99, 2007.

[24] O. Sukwong, H. S. Kim, and J. C. Hoe. Commercial antivirus software effectiveness: An empirical study. *IEEE Computer*, 44(3):63–70, 2011.

[25] R. Villamarín-Salomón and J. C. Brustoloni. Using reinforcement to strengthen users' secure behaviors. In *CHI*, pages 363–372, 2010.

[26] Y. Wang, R. Agrawal, and B.-Y. Choi. Light weight anti-phishing with user whitelisting in a web browser. In *Proceedings of IEEE Region 5 Conference*, pages 1–4, Apr. 2008.

[27] R. Wash. Folk models of home computer security. In *SOUPS '10*, pages 11:1–11:16, New York, NY, USA, 2010. ACM.

[28] Y. Wu and R. H. C. Yap. Towards a binary integrity system for windows. In *ASIACCS '11*, pages 503–507, New York, NY, USA, 2011. ACM.

[29] G. Wurster and P. C. van Oorschot. Self-signed executables: restricting replacement of program binaries by malware. In *Proceedings of the 2nd USENIX workshop on Hot topics in security*, pages 8:1–8:5, Berkeley, CA, USA, 2007. USENIX Association.

[30] H. Xia and J. C. Brustoloni. Hardening web browsers against man-in-the-middle and eavesdropping attacks. In *WWW*, pages 489–498, 2005.

Using Automated Model Analysis for Reasoning about Security of Web Protocols

Apurva Kumar
IBM Research - India
4 Block C, Institutional Area
Vasant Kunj, New Delhi, India-110070
kapurva@in.ibm.com

ABSTRACT

Interoperable identity and trust management infrastructure plays an important role in enabling integrations in cloud computing environments. In the past decade or so, several web-based workflows have emerged as de-facto standards for user identity and resource access across enterprises. Establishing correctness of such web protocols is of immense importance to a large number of common business transactions on the web. In this paper, we propose a framework for analyzing security in web protocols. A novel aspect of our proposal is bringing together two contrasting styles used for security protocol analysis. We use the *inference construction* style, in which the well-known BAN logic has been extended to reason about web protocols, in conjunction with, an *attack construction* style that performs SAT based model-checking to rule out certain active attacks. The result is an analysis method that shares simplicity and intuitive appeal of belief logics, at the same time covers a wider range of protocols, along with an ability to automatically find attacks. To illustrate effectiveness, case study of a leading web identity and access management protocol is presented, where application of our analysis method results in a previously unreported attack being identified.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General– *Security and protection*; C.2.6 [**Computer-Communication Networks**]: Internetworking Standards; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection.

General Terms

Security, Verification

Keywords

Security Protocols, Belief Logic, Automated security analysis

1. INTRODUCTION

Analysis of cryptographic protocols (i.e. protocols that use cryptographic techniques for distributing keys and authenticating principals over a network) has been an active area of research over the past three decades. Even seemingly simple protocols have the reputation of being notoriously error-prone [26] when exposed to an environment where the intruder is allowed to intercept, alter, delete messages and collude with dishonest principals.

In the last decade or so, a new set of protocols has emerged that manage specific transactions on the web. The protocols are characterized by a user interacting with multiple collaborating service providers, using standard web security mechanisms over a web-browser. Examples include cross-domain web single sign-on, secure electronic payments, content sharing with third parties etc. Industrial protocols implementing these transactions are responsible for security of cross-domain collaborations on the web. Some popular web protocols that have been used to implement such transactions are Security Assertion Markup Language (SAML) [2], OpenID [3] and OAuth [4].

Approaches for security protocol analysis can be broadly classified under two categories. Inference construction approaches, first popularized by the publication of Burrows, Abadi, Needham (BAN) [1] logic, attempt to use inference in specialized logics to establish required beliefs at protocol participants. These approaches have the advantage of resulting in efficiently computable formulations. Attack construction approaches, on the other hand, use model checking tools to construct attacks by modeling an intruder and using algebraic properties of the messages being transmitted. These complexities result in such approaches suffering from state-space explosion problem. We feel that inference construction based approaches (also termed as belief logics) are ideally suited for analyzing security of web protocols. The higher abstraction level and their ability to establish what a protocol achieves (in terms of beliefs established at participants) make them attractive for analyzing security in complex web transactions.

Belief logics, however, suffer from a few significant limitations. Firstly, soundness has been challenged due to their inability of handling certain types of active attacks. In particular, protocols suffering from attacks utilizing multiple parallel executions have been declared safe using these methods. Secondly, unlike model-checking, these approaches do not automatically generate an attack trace when expected beliefs cannot be established. Finally, while there have been several extensions of the original BAN, very little work has been done in extending these approaches to the web domain and supporting new types of attacks that are introduced due to browser-based communication.

In this paper, we propose a generic approach for analyzing web protocols. We perform security analysis in two stages. At a higher abstraction level, we perform inference construction using a logic that extends BAN to facilitate reasoning about web protocols. The

ACSAC '12, Dec. 3–7, 2012, Orlando, Florida, USA.

reasoning makes underlying assumptions about some security properties. At a lower abstraction level, we use a model finder to determine if attacks violating the property assumed by the belief logic are allowed by the protocol.

While there have been several extensions to BAN, our logic generalizes some basic concepts of the logic, towards supporting analysis of web protocols. We recognize the need to support users without identifying keys and having identities that are not global. To further simplify analysis for the web, we introduce a primitive construct in our logic to represent an SSL/TLS [29] based secure session. We also provide a framework for representing and reasoning about user actions. Users contribute in these protocols through actions like submitting a form, signing in, accepting terms, clicking a link etc. When identities are not global, establishing that a user recently performed an action is often more important than knowing its identity.

We identify an important security property which ensures that several session based attacks can be ruled out, thus allowing the belief logic analysis to be sound. To validate the security property, we develop a general model for web protocols in Alloy [15], a SAT based model analysis tool. Checking of this property is done for a much simplified version of the original protocol and a very simple intruder model, thus drastically reducing the overall complexity of our approach in comparison with typical attack construction approaches.

We illustrate effectiveness of our approach through security analysis of the OAuth protocol, an industry standard for web-based third party delegation. The analysis illustrates how our approach allows typical advantages of belief logics to be extended to web protocols. At the same time, we demonstrate that at a marginal overhead, we get benefits of model checking approaches like coverage against a wider range of attacks and automatic generation of attack traces.

We discuss related work in Section 2 and overview of BAN logic in Section 3. In Section 4, we introduce syntax and inference rules of the proposed logic. In Section 5, we introduce the hybrid approach that uses model analysis in conjunction with the logic. Section 6 describes Alloy based web protocol modeling in detail. An example analysis of OAuth is presented in Section 7, while important contributions of the work are covered in Section 8.

2. RELATED WORK

In the previous section, we mentioned two types of approaches for security protocol analysis. In this section, we review existing work in each type of approach.

Inference construction approaches attempt to use inference in specialized logics to establish required beliefs at the protocol participants. The logic of authentication described in [1], commonly known as BAN, was one of the first successful attempts at representing and reasoning about security properties of protocols. In [6], minor improvements to the logic's syntax and inference rules suggested to remove some ambiguity. Authors of [7] introduced the concept of 'recognizability'. Logic in [5] introduces the concept of possession along with belief and uses it to support constructs like 'not originated here'. In [8] authors attempt to consolidate good features from earlier belief logic approaches. These logics have the advantage of being usually decidable and efficiently computable. The logics can be easily automated. In [9], a transformation of BAN logic and inference rules to first order formula is performed and theorem prover SETHEO is used for finding proofs. In [10], the authors attempt to embed BAN logic in EVES theorem prover. However, given that a real protocol has a limited number of keys, principals and messages, forward chaining approaches discussed in [28] or the model driven analysis approach in [23] are often much simpler.

Attack construction approaches on the other hand do not try to establish beliefs at the participants but use model-checking techniques to construct attacks. The states and transitions used for modeling the protocol include modeling the structure of the message passing over the channel and a model of the intruder. The intruder is usually based on a Dolev-Yao model [11], and is allowed to perform any sequence of operations such as data interception, concatenation, de-concatenation, encryption, decryption etc. These complexities result in such approaches suffering from state-space explosion problem. However, these approaches do have the advantage of generating a counter-example corresponding to the attack, when a security property is not satisfied.

Few works that are representative of this class of approaches are mentioned below. The first such approach was introduced in [11], but the class of protocols studied in this work was very limited. In [12] the author modeled an extension of Dolev-Yao model in a specialized Prolog based model-checker, the NRL protocol analyzer. Other approaches in this area include the use of FDR model checker for CSP [13], use of SAT based model-checking techniques to solve a simplified version of the protocol insecurity problem [14] and on-the-fly model-checker (OFMC) [18], a semi-decision procedure which explores the search space system in a demand-driven way. [14] and [18] have been employed as backend model-checkers in the AVISPA tool [19] for automated validation of security protocols. The Proverif tool [30] is based on replacing the more accurate multi-set rewriting representation with abstractions that allow it to perform unbounded analysis of small to medium sized protocols. It uses an extension of applied-pi calculus as its input language. An alternative to state-based analysis is the strand-space based approach [20] which uses a graph-theoretic interpretation of Dolev-Yao model. The protocol analyzer, Athena [21] and the more recent Scyther tool [31] are based on this approach.

Since our work has elements of both attack and inference construction approaches, we now compare and contrast our work with most relevant works in each category. [23] possibly represents a first attempt at a belief logic for the web. The preliminary logic was able to analyze simple web protocols correctly. The main drawback was not being able to handle browser-based attacks e.g. cross-site request forgery. In this work, we improve the logic and augment the belief logic analysis with model checking methods. We analyze a version of OAuth different from the one analyzed in [23]. This version suffers from an attack that cannot be handled by the logic of [23].

Authors of [27] model a non-compliant version of the SAML single sign-on protocol and a standard Dolev-Yao adversary using multi-set rewrite formalism and discover an attack resulting from insufficient authentication between service providers. However, use of the standard adversary instead of a specialized browser based attacker, results in another flaw, similar to the one identified in [22] for WebAuth protocol, remaining unexposed.

Authors of [22] model a few web mechanisms using Alloy [15] and analyze them for multiple security properties. They also show how to analyze simple web protocols using their framework. For the second stage of our analysis, we also develop a generic model

in Alloy, but we are able to handle much more complex protocols since we use model analysis in conjunction with belief analysis. This results in Alloy based analysis being performed for a significantly simplified version of the original protocol.

3. OVERVIEW OF BAN

BAN statements. A formula in BAN logic [1] is constructed using operators from Table 1. *P* and *Q* range over principals. The three statements about keys and secrets represent atomic statements. *X* represents a BAN formula constructed using one or more BAN operators. The expression $\sharp X$ means that the message *X* is fresh and has not been used before the current run of the protocol. This is especially true for a nonce, a sequence number or timestamp generated with this specific purpose. Nonces are used in protocols to defeat replay attacks from previous executions of the protocol. The *said* and *freshness* operators can be combined into a single *says* operator.

Table 1. Operators in BAN Logic. *X* is a statement of the logic.

Notation	Meaning	Notation	Meaning
$P \mid\equiv X$	*P* believes *X*	$P \overset{K}{\leftrightarrow} Q$	Shared key *K*
$P \lhd X$	*P* sees *X*	$\mapsto_K Q$	*Q* has public key *K*
$P \mid\sim X$	*P* said *X*	$P \overset{Y}{\rightleftharpoons} Q$	Shared secret *Y*
$P \mid\Rightarrow X$	*P* controls *X*	$\sharp X$	*X* is fresh.
$\{X\}_K$	*X* encrypted by *K*	$\langle X \rangle_Y$	*X* combined with *Y*

Inference Rules. There is a set of inference rules for deriving new beliefs from old ones. E.g. the *message-origin* inference rule below states that if *P* knows that *K* is a secret key between itself and *Q* and it sees a message *X* encrypted by *K*, then *P* is entitled to believe that *Q* said *X*. Similar inference rules about public keys and shared secrets are also provided, as shown below. K^{-1} represents the private key corresponding to public key *K*.

$$\frac{P \mid\equiv Q \overset{K}{\leftrightarrow} P, P \lhd \{X\}_K}{P \mid\equiv Q \mid\sim X} \qquad \frac{P \mid\equiv \mapsto_K Q, P \lhd \{X\}_{K^{-1}}}{P \mid\equiv Q \mid\sim X}$$

$$\frac{P \mid\equiv Q \overset{Y}{\rightleftharpoons} P, P \lhd \langle X \rangle_Y}{P \mid\equiv Q \mid\sim X} \qquad \text{(R1)}$$

A *nonce-verification* rule (R2) states that, in addition if the message is known to be fresh, then *P* believes that *Q* must still believe *X*. Further, the *jurisdiction rule* (R3) states that, if in addition, *P* also believes that *Q* is an authority on the subject of *X* (i.e. *Q* controls *X*), then *P* is entitled to believe *X* itself.

$$\frac{P \mid\equiv Q \mid\sim X, P \mid\equiv \sharp X}{P \mid\equiv Q \mid\equiv X} \quad \text{(R2)} \qquad \frac{P \mid\equiv Q \mid\equiv X, P \mid\equiv Q \mid\Rightarrow X}{P \mid\equiv X} \quad \text{(R3)}$$

Idealization. Each message exchanged in the protocol is idealized into a BAN formula representing meaning of the message including any facts that the sending of the message implies. Consider for example, the second message in the Needham-Schroeder Symmetric Key protocol in which a server *S* sends a response to an initiator *A* containing a session key K_{ab}, along with a message for another principal *B* encrypted using *B*'s key containing the same session key and *A*'s identity. In typical Alice-Bob notation used in literature this can be expressed as:

$$S \rightarrow A : \{N_a, B, K_{ab}, \{K_{ab}, A\}_{K_{bs}}\}_{K_{as}}$$

where N_a is a nonce value. K_{as} and K_{bs} represent keys shared between *A* and *S*, *B* and *S* respectively. The message is idealized in [1] as follows:

$$S \rightarrow A : \{N_a, (A \overset{K_{ab}}{\leftrightarrow} B), \sharp(A \overset{K_{ab}}{\leftrightarrow} B), \{A \overset{K_{ab}}{\leftrightarrow} B\}_{K_{bs}}\}_{K_{as}}$$

The idealization makes explicit that the server says that K_{ab} is a shared key for communication between *A* and *B* and also that it is fresh (due to the presence of the nonce).

Analysis. Protocol analysis in inference construction approaches involves two main tasks: (*i*) identification of an initial set of beliefs i.e. assumptions at each principal. (*ii*) message-by-message manual reasoning based on combining formula (idealized messages) that a principal sees with what it knows using inference rules of the logic.

4. BELIEF LOGIC FOR WEB PROTOCOLS

4.1 Need for Extending Belief Logics

4.1.1 Typical Web-based Workflow

In a typical web-based protocol workflow, a user interacts with web-pages presented to him by one or more providers by performing actions through a user-agent (web-browser). Examples of actions are: accessing a service by clicking on a link, submitting a form, signing in, agreeing to terms etc. Since interaction take place over the stateless HTTP protocol, application state is encoded in secrets (usually an HTTP header field called cookie) and returned to the user in response. Considering that user actions are responsible for state transitions, secrets are associated with specific user actions. Secrets are usually transferred over secure SSL/TLS channels.

In workflows involving multiple providers, the user-agent processes an HTTP redirect response requesting user to be transferred from one provider domain to another. In such collaborations, a degree of trust exists between the providers and tokens, secrets issued to a user in one domain may be acceptable in another domain. Such secrets are included in the body of an HTTP request rather than a cookie header since cookies are domain specific. With this context, we now highlight some significant departures from typical cryptographic protocols that motivate the need for a belief logic designed for the web.

4.1.2 Principals without Identifying Keys

Cryptographic protocols assume that principals possess identifying keys, either a private key (public key cryptography) or a key shared by an authentication server (shared key cryptography). Identities associated with keys are global. While service providers on the web often possess identifying keys issued by trusted authorities, end-users of web protocols typically do not possess identifying keys. Moreover, when protocols require identifying a user by name, identities established are local to a provider. In web protocols, it is also not uncommon to uniquely identify a user through an action it performs rather than a name. For these reasons, we allow principals without identifying keys:

(*a*) a user that has recently performed a sign-in action (and not signed out yet) at a provider is considered a principal with a named local identity, (*b*) a principal can also be identified by virtue of any other protocol specific action it performs, e.g. a user who recently registered at a website with certain data items.

4.1.3 Need to Model Secure Channels

In the absence of identifying keys, some other mechanism is required to associate messages with principals. Clear-text communication is ignored by belief logics since it does not identify the source. However, most web transactions make use of underlying SSL/TLS based secure channels that provide unilateral (server) authentication, confidentiality and integrity. Since security properties of transport layer security mechanisms are well understood, it greatly simplifies security analysis if these properties are assumed rather than proven for each occurrence a secure channel. Secure channels not only allow associating statements with principals without identifying keys but also ensure fixed end-points and freshness of message exchanges.

4.1.4 Goals for Web Protocols

In a cryptographic protocol, important beliefs generated at principals are those concerning appropriateness of keys and secrets. In web protocols, secrets and keys used are transparent to the user and thus rarely appear as protocol goals. Instead goals are often expressed as belief about a user or principal performing an action. E.g. ensuring that user accessing a service, is the one who completed a successful payment for the service, is a reasonable goal for a web protocol.

4.2 Extended Syntax

We introduce two new types of objects (sorts) in the logic: *user* and *action*. A *user* is defined as the client side of a secure channel. We use the channel identifier as a subscript in our notation for user. We note that 'user' in our logic is a concept distinct from 'principal'. A user (end-point of a secure channel) could be signed in as different principals at different epochs within the same SSL/TLS session. In the rest of the paper, depending on the context, the term 'user' refers either to a specific instance of the sort user or a principal playing the end user role.

For a given protocol, we define a set of function symbols *Atype* - each associated with an arity of the form $\sigma_1 \times \ldots \times \sigma_n \rightarrow$ action - representing the type of actions in the protocol, where $\sigma_1, \ldots \sigma_n$ are other sorts in the logic. Then $Aname(v_{\sigma 1}, \ldots, v_{\sigma n})$, ($Aname \in Atype$), represents an instance of an action type executed in a protocol run (with argument $v_{\sigma i}$ representing a constant or variable of sort σ_i). Actions could either be generic or specific to a particular application. E.g. signing in as principal *Q*, represented as *SignIn*(*Q*) is a generic action.

Table 2. Additional operators used in our extended logic.

$P \overset{\Delta}{\longleftrightarrow} U_C$: *C* is a secure channel between user U_C and *P*.
$[\![X]\!]_C$: Formula *X* exchanged over secure channel *C*.
$U_C \ni X$: User U_C possesses secret X.
$X \rightsquigarrow Aname(v_{\sigma_1}, \ldots, v_{\sigma_n})$: Secret *X* associated with action.
$U_C \triangleright Aname(v_{\sigma_1}, \ldots, v_{\sigma_n})$: User U_C performs action.

4.3 Inference Rules

We introduce inference rules of the logic that allow simple reasoning about users, their actions and associated secrets. R4.1 says that if a principal *P* (usually server) believes that a user U_C is communicating over a secure channel *C*, then any actions it sees over the secure channel *C* can be attributed to user U_C. According to R4.2, any tokens seen over a secure channel are assumed to be possessed by the user. R4.3 says that when the client side receives a statement *X* over a secure channel, it is entitled to believe that the server principal believes *X*.

$$\frac{P \models (P \overset{\Delta}{\longleftrightarrow} U_C), P \lhd [\![action]\!]_C}{P \models (U_C \rhd action)} \quad (4.1) \qquad \frac{P \models (P \overset{\Delta}{\longleftrightarrow} U_C), P \lhd [\![X]\!]_C}{P \models (U_C \ni X)} \quad (4.2)$$

$$\frac{U_C \models (P \overset{\Delta}{\longleftrightarrow} U_C), U_C \lhd [\![X]\!]_C}{U_C \models P \models X} \quad (4.3) \qquad (R4)$$

R4.1 requires the believing principal to be a direct observer of the action. R5, on the other hand, allows belief about an action based on a secret associated with the action. If *P* believes secret *S* to be associated with *action*, and it sees user U_C possessing the secret, then it believes that *action* was performed by U_C

$$\frac{P \models (S \rightsquigarrow action), P \models U_C \ni S}{P \models (U_C \rhd action)} \qquad (R5)$$

While R4.1, R5 do not require a user to be authenticated, R6 is the corresponding rule for authenticated users. R5 says that if *P* is connected to U_C over a secure channel and believes that U_C is currently signed in as *Q*, then *P* can attribute any actions seen on the channel to the principal *Q*. We use the predicate *SignedIn*(U_C, *Q*) to denote that U_C is signed in as *Q*.

$$\frac{P \models (P \overset{\Delta}{\longleftrightarrow} U_C), P \models SignedIn(U_C, Q), P \lhd [\![action]\!]_C}{P \models (Q \rhd action)} \qquad (R6)$$

For browser based protocols, presence of a cookie can be used to establish whether a principal is signed-in or is aware of an action associated with a cookie. We use variable name *ck* and constants named Ck_x (*x* is a principal name or an action) for cookies.

$$\frac{P \models (P \overset{\Delta}{\longleftrightarrow} U_C), P \models ck \rightsquigarrow SignIn(Q), P \lhd [\![ck]\!]_C}{P \models SignedIn(U_C, Q)} \qquad (R7)$$

Due to space limitation, we are unable to include a proof of soundness but walk through important steps here. The proof builds on semantics of BAN described in [6]. In addition to the model of [6], we associate a user with two collections: a sequence of actions executed by it and a set of actions it is allowed to assert based on possession of secrets. We introduce semantics for secure channel and use it to establish soundness of R4. Soundness of R5, R6 is established by proving if premises are satisfied then (*i*) an action is known to appear in one of the collections (semantics of user performing action), or (*ii*) an action appears in the sequence after a *SignIn* action (semantics of principal performing action).

We note that our semantics assumes a user to be *aware* of actions it is allowed to assert. This is not always true for web protocols. In presence of cross-site request forgery where secrets are forged, the assumption is not valid. A web protocol negates such attacks if it satisfies the redirection safety property described in Section 5.4.

5. REASONING BASED ON SECRETS IN WEB PROTOCOLS

5.1 Attacks Based on Parallel Runs

A significant limitation of belief logics has been their inability to handle certain attacks based on multiple protocol runs. In such attacks, intruder participates in multiple simultaneous executions of the protocol and uses secret values learnt from one session, in another session. A case in point is the multiple session based attack on Needham-Schroeder Public Key (NSPK) protocol first reported in [13]. The protocol was analyzed as safe in the original BAN work [1]. While this does not impact BAN's reasoning based on signed messages, it does demonstrate an inadequacy related to reasoning based on secrets.

5.2 Request Forgery in Web Protocols

In web protocols, an additional complexity introduced while dealing with secrets in is that of request forgery. In cryptographic protocols, a principal is always aware of the content of messages it sends (unless it is relaying an encrypted message for which it is not held accountable). In browser-based protocols, a user may be induced into clicking a link or submitting a form at a malicious website. Both the content of the message and the receiving end-point can thus be controlled by an attacker. Moreover, an HTTP cookie identifying any context information (e.g. login context) is automatically inserted by the user-agent if the request is directed to a URL within its scope (defined as a combination of domain and path). The class of attacks is termed as cross-site request forgery (CSRF). Since secrets are message content and can be forged, reasoning in a logic designed for the web should include measures to ensure soundness in presence of such attacks.

5.3 Using Model Checking to Supplement Belief Logic Analysis

One way to address issues described in Sections 5.1 and 5.2 is to extend the logic to address the above scenarios. However, not only will this make the logic more complex, it goes against the spirit of inference construction approaches to build support for specific attacks. Thus, instead of extending the logic, we take the approach of augmenting belief logic analysis with attack construction approaches. The general approach works as follows. We use the belief logic to analyze the protocol first and obtain a set of beliefs established by the protocol. However, these beliefs are subject to a security property being satisfied by the protocol. The specific security property can then be checked using a model checking approach. Moreover, depending on the security property, it may be possible to use a simplified version of the protocol, to check for it.

In Section 5.4, we discuss such a security property for web protocols. The belief logic of Section 4 is designed to be used with the assumption of this property. In Section 5.5, we show that validation of the property can be performed using a much simplified version of the original protocol being analyzed.

5.4 Secure Redirection Property

Secrets are used in web protocols primarily to identify a user who has performed an action, possibly at a different place or time. Secrets can either be included in the HTTP header as a cookie, or carried in the body of the message. In the presence of request forgery, secrets carried in the body of the request cannot by themselves be used to conclusively establish beliefs at service providers.

Assumption of secure channel (HTTPS) makes protocols easier to analyze. Since most protocols carrying secret information recommend using secure HTTPS communication, the assumption is not unrealistic. An SSL/TLS session eliminates a major class of man-in-the-middle attacks. With secured sessions, in order to forge requests containing secrets, the adversary is required to be a participant in the protocol. Since authentication of service providers is analyzed using belief logic in our approach, the model checking stage needs to only rule out possibility of a malicious third party participating as a user in the protocol. Moreover, malicious user behavior e.g. tricking an honest user into joining a manipulated protocol session can only occur between secure sessions. The intruder model is thus greatly simplified. In particular, presence of an adversary, is indicated by a 'message' exchanged between two users (one honest and one dishonest). The communication could happen through an asynchronous user action e.g. clicking a malicious hyperlink sent by the attacker, or an automated browser action e.g. redirection to a manipulated callback URI. If we can establish that the protocol does not allow user-to-user communication, we say that it satisfies the secure redirection property.

5.5 Protocol Simplification

Establishing security property of Section 5.4 essentially involves associating two or more secure sessions in the protocol with the same web user. In the absence of identifying keys, possession of secrets and cookies is the only means to establish this. Since this analysis only requires reasoning based on possession of secrets and cookies, it can be done on a much simplified version of the protocol in which only messages containing secret values exchanged with a user are retained. The rules for simplification described below are tabulated in Table 3.

Rule 1: A protocol message that is not sent or received by a user can be removed, i.e. server to server communication is ignored.

Rule 2: Any term which does not contain a secret or nonce value is dropped.

Rule 3: In the remaining messages, an encrypted formula $\{X\}_K$, received at a user is represented by an opaque token $N_{x,k}$ if the decryption key can be assumed to be unavailable to the intruder. Otherwise, in addition to the token, the formula is also included (with or without encryption as shown in Table 3).

Rule 4: Similarly, an encrypted formula $\{X\}_K$ received from a user is interpreted as the user possessing the corresponding token $N_{x,k}$

Table 3. Transformation rules for sent, received messages

Formula	Decryption Key in Intruder Knowledge	Transformed Formula	
		Server to User	User to Server
$\{X\}_{K^{-1}}$	Always	$X, N_{x,K^{-1}}$	$N_{x,K^{-1}}$
$\{X\}_K$	Never	$N_{x,K}$	$N_{x,K}$
$\{X\}_{Ksh}$	Always	$X, N_{x,Ksh}$	$N_{x,Ksh}$
$\{X\}_{Ksh}$	Never	$N_{x,Ksh}$	$N_{x,Ksh}$
$\{X\}_{Ksh}$	No assumption	$\{X\}_{Ksh}, N_{x,Ksh}$	$N_{x,Ksh}$

K: public key, K^{-1}: private key, K_{sh}: shared key.

6. MODELING WEB PROTOCOLS USING ALLOY

6.1 Overview of Alloy

Alloy [15], [17], [29] is a declarative language for describing structures and a tool for exploring them. An alloy model specifies a set of constraints that apply to objects in the domain being modeled. Alloy Analyzer is a solver that takes constraints of a model and finds structures satisfying them using a SAT solver. Thus technically, it is a model-finder rather than a model-checker. A *signature* and a constraint on the signature are declared below:

```
sig S extends E {
      F: one T }
fact {
      all s:S | s.F in X }
```

It is often useful to think of Alloy as an object-oriented language, e.g. `S` is a class (`s` being an instance), that extends superclass `E`. `F` is a member of `S` pointing to `T`. However, under the covers `S` is a subset of `E` and `F` is a relation that maps each of `S` to a single `T`.

Fact statements represent constraints that must always hold. Quantified expressions of the form *quantifier* `s: S | F` mean that constraint F holds for `all`, `no`, `lone` (zero or one), `some` (at least one) or `one` elements of S. Fact expressions that apply to a particular signature (as is the case above) can be directly appended to the signature within curly brackets. Predicate (`pred` {…}) and functions are optional facts that can be conditionally invoked. Assertions (`assert` {…}) are properties against which the specification needs to be checked. A `run` command causes the analyzer to search for consistency of a function or predicate, while a `check` command causes it to search for a counter-example to show that the assertion does not hold.

Alloy checks models of finite sizes using a specified scope which limits the maximum size of top level signatures. Exceptions can be specified for specific signatures. E.g. the below command tries to find a counter-example for assertion `acyclic` with default scope up to 5, but up to 2 `fileSystem`s and 7 `FSObject`s.

```
check acyclic for 5 but 2 fileSystem,
exactly 7 FSObject
```

We also use a utility `ordering` to define an order on elements of a signature. The function greater than or equal (`gte`) defined in `ordering` can be used in an expression such as the one shown in example below to specify `vals` with times ≥ `time`.

```
open util/ordering[Time] as ord
..
all t: Time |t in vals <=> t.ord/gte[time]
```

6.2 General Model for Web Protocols

We now describe our general Alloy model that allows reasoning about secrets for a wide range of web protocols.

Principals. The signature `Process` declares a set of all principals. It is extended by signatures `Server` and `User` which are (disjoint) subsets representing web service providers and end users respectively, Also declared are set of all keys (`Key`), private keys (`PvtKey`), instants (`Time`), cookies (`Cookie`) and values, (`Value`, `CkValue`, `TkValue`).

A private key is associated with a public key through the relation `pubkey`. A principal knows a set of keys (`knownkeys`) and a server principal owns a private key (`ownedkey`). Most web protocols requiring security analysis involve two collaborating service providers. The `peer` relation maps a server to its collaborating partner. The relations `uniquecookie` and `uniqueval` associate a `Server` with a unique cookie and a secret/nonce value, respectively. Minor changes in the declarations are required to represent protocols needing more than cookie, secret per server role. Constraint on `uniquecookie` relation ensures that cookie points to the correct server.

```
abstract sig Process {
   knownkeys: set Key }
sig Time { }  sig Key { }   sig Value { }
sig TkValue extends Value { }
sig CkValue extends Value { }

sig Cookie{
    value: one CkValue, server: one Server }
sig PvtKey extends Key {
    pubkey: one Key } { pubkey != this }
sig Server extends Process {
    ownedkey: one PvtKey,peer: one Server,
    uniqueval: one TkValue,
    uniquecookie: one Cookie
} {peer != this, uniquecookie.server = this}

sig User extends Process {
    seentokens: set TkValue->Time,
    knowncookies: set Cookie->Time}{ ... }
sig HUser extends User { }

fact {all k1,k2: PvtKey|k1 != k2 =>
k1.pubkey != k2.pubkey }

fact { all s1,s2: Server|s1 != s2 =>
(s1.uniqueval != s2.uniqueval) &&
(s1.ownedkey != s2.ownedkey) &&
(s1.uniquecookie)!= (s2.uniquecookie) }
```

`User` participates in two relations. `seentokens` associates the user to a set of (*value*, *time*) pairs each indicating that a *value* was known to user at *time*. The relation `knowncookies` provides a similar association for cookies known to the user.

Finally, the two facts represent constraints on private keys and servers. Two private keys may not be mapped to the same public key. Similarly each server must own a unique private key and should generate distinct nonces and cookies.

Protocol Messages. The signature `Sent` is used to declare a set of possible protocol (HTTP) messages. Each message has a `sender` and `receiver` principal and is associated with a `time` when it is transmitted. The other relations on message are a set of values (`content`) and a set of cookies (`cookies`) contained in the message. A message may also contain a redirection URL (`redirectURL`), if it represents an HTTP redirect. If a key is used to encrypt the message it is identified using the `enckey` relation. The first constraint says that a message sent by a user can only contain cookies that are known to sender at the time of sending the message and were received earlier from the target of that message. The bi-implication requires that all such cookies must necessarily be included in the message.

```
sig URL { target: one Process }
sig Sent {
   cookies: set Cookie, sender: one Process,
   receiver: one Process, time: one Time,
   content: set TkValue,
   redirectURL: lone URL, enckey: lone Key }
{
  sender in User => (all c: Cookie | (c in
  cookies <=> c->time in sender.knowncookies
  && c.server = receiver))
  sender in User => (all v: TkValue | (v in
  content => v->time in sender.seentokens))
  enckey in sender.knownkeys+sender.ownedkey
  sender != receiver }
```

The next one is a similar constraint requiring any values contained in a message to be known to the user. The third constraint requires the encryption key to be known to the sender. The last constraint says that sender and receiver of a message have to be distinct. More complex messages can be modeled by replacing `content` with an appropriate structure.

Learning Rules. The rules for a user learning new secret values or cookies are expressed as constraints appended to the `User` signature. An ordering is defined on the elements of `Time`. The first constraint implies that a pair (`cookie`, `t`) appears in `knowncookies` if and only if the user has seen a message containing `cookie` at a time ≤ `t`. A similar constraint for values seen by a user is also specified.

```
open util/ordering[Time] as ord

sig User extends Process { ...} {

   all c: Cookie | all t: Time | (c->t in
knowncookies <=> some s: Sent | c in
s.cookies && s.receiver = this &&
t.ord/gte[s.time])

   all v: TkValue | all t: Time | (v->t in
seentokens <=> some s: Sent | v in s.content
&& s.receiver = this && s.enckey in
s.reciver.knownkeys && t.ord/gte[s.time]) }
```

Protocol Flow. The signature `ProtoSeq` represents all possible sequences of messages under generic and protocol specific constraints.

```
sig ProtoSeq {
   sequence: set Sent->Sent
} {
   all p,q: Sent | (p->q in sequence) =>
(q.sender = p.receiver)
   all p,q: Sent | (p->q in sequence) =>
(q.time = ord/next[p.time])
   all p: Sent | (p.receiver in HUser) &&
p.redirectURL => (some q: Sent | (p->q in
sequence) && (q.receiver =
p.redirectURL.target) && (q.content =
p.content))
 /* other protocol specific constraints */ }
```

If `p` and `q` are possible sent messages, then `p->q` appearing in the sequence implies that receiver of `p` is the sender of `q`. Also the timestamp on `q` must be the next time instant following the timestamp of `p`. The last generic constraint describes handling of an HTTP redirect for an honest user (`HUser`). It specifies that if an honest user receives a redirect message, the next message in the sequence must be a message sent by this user to the target of the redirection URL. The message should include any values/tokens received in the redirect. The other constraints on protocol sequence are specific to the protocol being modeled.

Intruder Model. The intruder is simply a `User`. The redirection constraint for honest user does not apply to it. The intruder learns new values based on learning rules for tokens and can only send seen tokens (as per constraint on `Sent` discussed earlier). Communication from a dishonest to honest user is modeled as a redirect message generated by the dishonest user.

7. ANALYSIS OF OAUTH PROTOCOL

7.1 Protocol Description

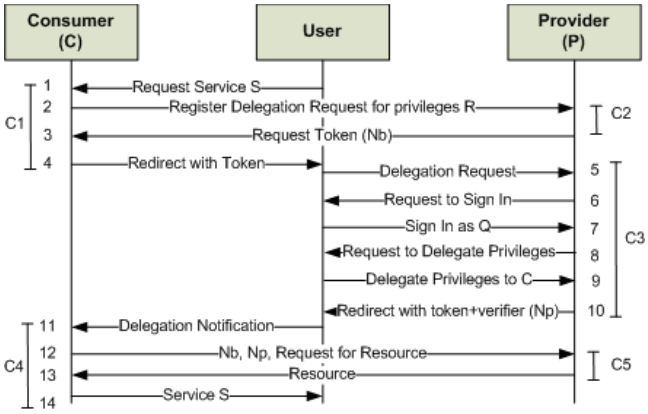


Figure 1. The OAuth Protocol (RFC version)

The OAuth protocol [4] provides a web based workflow that allows a user to temporarily delegate privileges of his account at a provider to a third party without sharing his login credentials. Privileges could for example mean access to pictures, friend list, blogs etc. OAuth is the primary protocol used by Google, Facebook, Twitter to allow third party access to user content.

The original version, OAuth Core 1.0 [24] is known to suffer from a session-fixation attack and was analyzed in [23]. In this paper, we analyze the revised version (Core 1.0a) also an approved IETF RFC [4], *OAuth 1.0 Protocol*. We also refer to this as the *RFC version* of OAuth[1]. Workflow for the RFC version of OAuth protocol shown in Figure 1 is described below.

Steps 1-4, user requests service *S* from consumer (*C*). The service requires a set of privileges (permissions) *Priv* to the user account at provider (*P*). Consumer registers delegation request with *P* and gets returned a request token N_b. *C* redirects user to *P* with this token. *Steps* 5-10, user is requested to sign in and delegate set of privileges *Priv* to *C*. User signs in as principal *Q* and performs

[1] We note that some providers like Google, Facebook have moved to OAuth 2.0 [25] which bears little resemblance with the original protocol and is not analyzed here. Other providers e.g. Twitter have chosen to stay with the IETF approved version [4].

requested delegation. User is redirected back to *C* with the request token and another *verifier* token, N_p. *Steps 11-14*, *C* uses tokens N_b, N_p to request a protected resource directly from *P*. User receives requested service *S* in step 14. All communication happens over secure SSL/TLS channels and requests from consumer are signed and verifiable at the provider. The secure channels used are identified as C1-C5 in Figure 1.

7.2 Belief Logic Analysis of OAuth

The protocol is idealized as shown below. Only messages received by either *C* or *P* are idealized because we are interested in beliefs at these principals. There are two named variables in the idealized protocol: *scope* representing the set of privileges to be delegated and *callback* identifying the URL used by *P* for redirection in step 10.

$$\begin{aligned}
&Message\ 1 \quad U_{C1} \rightarrow C: \quad [\![Request(S)]\!]_{C1} \\
&Message\ 2 \quad C_{C2} \rightarrow P: \quad [\![\{N_1, scope = Priv, \\
&\qquad\qquad callback = url_c\}K_c^{-1}]\!]_{C2} \\
&Message\ 3 \quad P \rightarrow C_{C2}: \quad [\![N_b, callback = url_c, \\
&\qquad\qquad scope = Priv]\!]_{C2} \\
&Message\ 5 \quad U_{C3} \rightarrow P: \quad [\![N_b]\!]_{C3} \\
&Message\ 7 \quad U_{C3} \rightarrow P: \quad [\![SiginIn(Q)]\!]_{C3} \\
&Message\ 9 \quad U_{C3} \rightarrow P: \quad [\![Delegate(scope, C), Ck_Q]\!]_{C3} \\
&Message\ 11 \quad U_{C4} \rightarrow C: \quad [\![N_b, N_p]\!]_{C4} \\
&Message\ 12 \quad C_{C5} \rightarrow P: \quad [\![\{C, N_2, N_b, N_p, p1\}K_c^{-1}]\!]_{C5} \\
&Message\ 13 \quad P \rightarrow C_{C5}: \quad [\![N_p \rightsquigarrow Delegate(scope, C)]\!]_{C5}
\end{aligned}$$

Idealization of messages 1, 7 and 9 represents user actions performed in the protocol. Messages 2 and 12 are direct requests from *C* to *P* for request token and protected resource respectively. In message 2, the set of privileges *Priv*, for which delegation is required, is included. In message 12, a protected resource requiring a privilege $p1 \in Priv$ is requested. N_1 and N_2 are nonces (combination of a timestamp and nonce in actual protocol). Cookie Ck_Q represents login context for user signed in as principal *Q* at *P*. When *P* returns the requested resource in message 13, it conveys to *C* that the verifier token corresponds to a valid delegation action.

Apart from the more obvious assumptions about secure channels (C1-C5), knowledge of public keys, and freshness of nonces (N1 and N2), we also make the following assumptions.

$$\begin{aligned}
&C \models N_b \rightsquigarrow Request(S) && P \models N_p \rightsquigarrow Delegate(Priv, C) \\
&C \models (\forall x, r, P \models\Rightarrow x \rhd Delegate(r, C)) && P \models Ck_Q \rightsquigarrow SignIn(Q) \\
&C \models (\forall y, r, P \models\Rightarrow y \rightsquigarrow Delegate(r, C))
\end{aligned}$$

These include association of secrets N_b, N_p and cookie Ck_q with the user actions and *C*'s complete trust in *P* for the delegation action. The goal of the protocol is as specified below. (G1) ensures that the delegation was performed by the user who has previously performed the sign-in action at provider. (G2) is required to ensure that recipient of the service at Consumer in step 14 has also performed the delegation action.

$$P \models Q \rhd Delegate(priv, C) \qquad \text{(G1)}$$
$$C \models U_{C4} \rhd Delegate(priv, C) \qquad \text{(G2)}$$

We now present our belief logic analysis. The forward chaining based analysis can be easily automated using existing approaches like [28], [23] as mentioned in Section 2.

Message 2: OAuth token request received at *P*.

Combining received message with assumption about *C*'s public key using R1, followed by application of rule R2 using assumption about freshness of N_1.

$$\frac{P \lhd \{N_1, scope = \text{Priv}, callback = url_c\}K_c^{-1} \quad P \models \mapsto_{K_C} C \quad P \models \sharp N_1}{P \models C \models (scope = \text{Priv}, callback = url_c)} \text{(R1, R2)}$$

Message 9: Delegation Action seen by *P*.

First apply R7 to establish that U_{C3} is signed in as *Q* using presence of cookie and assumption about Ck_Q. Next, associate observed action with *Q* using R6.

$$\frac{P \models P \overset{\Delta}{\longleftrightarrow} U_{C3} \quad P \models Ck_Q \rightsquigarrow SignIn(Q) \quad P \lhd [\![Ck_Q]\!]_{C3}}{} \text{(R7)}$$

$$\frac{P \models SignedIn(U_{C3}, Q) \quad P \overset{\Delta}{\longleftrightarrow} U_{C3} \quad P \lhd [\![Delegate(Priv, C)]\!]_{C3}}{P \models [\![Q \rhd Delegate(Priv, C)]\!]_{C3}} \text{(R6)}$$

Message 11: Establish that $U_{C4} \ni N_b$ and $U_{C4} \ni N_p$ (R4.2).

Message 13: Use R4.3 to establish that *P* believes about validity of N_p and then use *C*'s trust in *P* for delegation action (R3). Finally, combine with conclusion of message 11 to infer that U_{C4} has performed the delegation action.

$$\frac{\dfrac{\dfrac{\dfrac{C \lhd [\![N_p \rightsquigarrow Delegate(scope, C)]\!]_{C5} \quad C \models P \overset{\Delta}{\longleftrightarrow} C_{C5}}{C \models P \models N_p \rightsquigarrow Delegate(scope, C)} \text{(R4.3)} \quad C \models P \models (scope = Priv)}{C \models P \models N_p \rightsquigarrow Delegate(Priv, C)}}{C \models (\forall y, r, P \models\Rightarrow y \rightsquigarrow Delegate(r, C))} \text{(R3)} \quad C \models N_p \rightsquigarrow Delegate(Priv, C) \quad C \models U_{C4} \ni N_p}{C \models U_{C4} \rhd Delegate(Priv, C)} \text{(R5)}$$

From analysis of messages 9 and 13, we find that goals G1 and G2 are satisfied. However, we still need to establish that the secure redirection property assumed by the belief logic holds for the protocol.

7.3 Checking Redirection Security using Alloy

In the protocol of Figure 1, messages 5 and 11 are the only messages demonstrating possession of secret with a user. The two secrets considered in the analysis are N_b and N_p. Applying protocol simplification rules of Section 5.5, we obtain the following simplified protocol. For each message, the message sequence number in the original protocol of Figure 1 is indicated in parentheses.

$$\begin{aligned}
&Message\ 1\ (4) \quad C \rightarrow U: \quad N_b, url_p \\
&Message\ 2\ (5) \quad U \rightarrow P: \quad N_b \\
&Message\ 3\ (10) \quad P \rightarrow U: \quad N_b, N_p, url_c \\
&Message\ 4\ (11) \quad U \rightarrow C: \quad N_b, N_p
\end{aligned}$$

Note that we assume that url_c points to *C* in our model due to the belief $P \models C \models callback = url_c$ established from analysis of message 2 in Section 7.2.

We use the general Alloy model described in Section 6.2. To add constraints corresponding to this protocol, we declare two new signatures `Producer` and `Consumer` by extending `Server`.

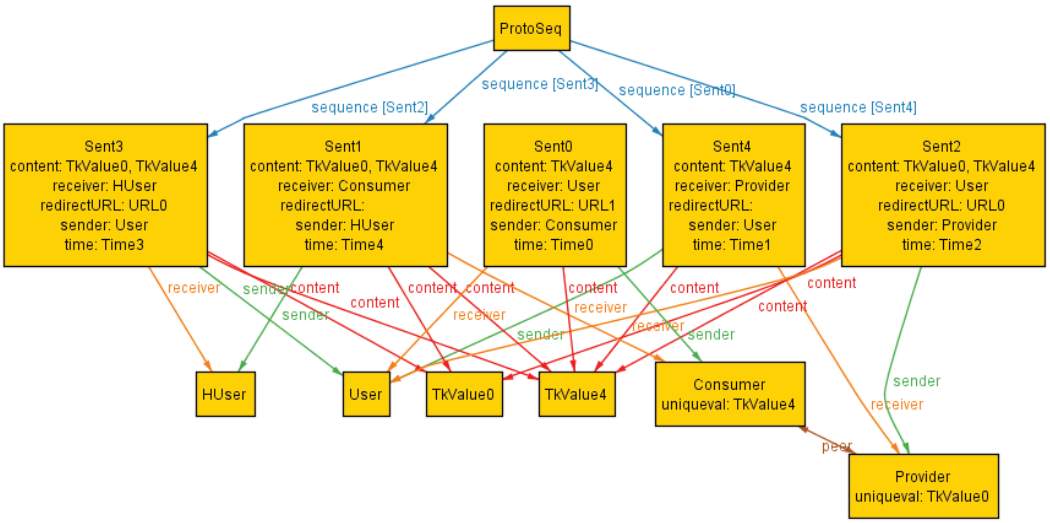


Figure 2. Counter-example generated by Alloy representing attack on *OAuth* (RFC version)

We also add following protocol specific constraints to the signature `ProtoSeq`:

Constraint 1: *P* must send message 3 containing verifier token and redirection URL in response to receiving message 2.

```
all p: Sent |  (p.receiver in Provider) =>
some q:Sent  | (p->q in sequence) &&
(q.receiver = p.sender) &&
(q.content=p.content + p.receiver.uniqueval)
&& (q.redirectURL.target =q.sender.peer)
```

Constraint 2: First message in protocol sequence must be sent by Consumer and must contain a secret (request token) and redirection URL pointing to the Producer.

```
one p: Sent | (some t1: Sent | p->t1 in
sequence && no t2: Sent | t2->p in sequence)
&& p.sender in Consumer && (p.content =
p.sender.uniqueval) && (p.cookies =
p.sender.uniquecookie) && (p.time =
ord/first[]) && (p.redirectURL.target =
p.sender.peer)
```

Constraint 3: Last message in the protocol must be received by the consumer and must contain request token and the verifier.

```
one p: Sent | (some t1: Sent | t1->p in
sequence && no t2: Sent | p->t2 in sequence)
&& p.receiver in Consumer && (p.content =
p.receiver.uniqueval +
p.receiver.peer.uniqueval)
```

We execute Alloy analyzer with these additional constraints and an assertion that there is only one `User` and it is honest (`HUser`). Alloy generates the counter example shown in Figure 2 in less than 4 seconds on an Intel Core i5 2.4 GHz, 4 GB system. The counter-example clearly shows possibility of two users, one of which is honest. Also there are five messages instead of expected four. The sequence is determined by the value of the time attribute. The value `Tkvalue4` represents the request token (N_b) while `TkValue0` is the verifier (N_p). Note that the redirection constraint of Section 6.2 only applies for an honest user (`HUser`). We see that a (dishonest) user can perform steps 1, 2, 3 of the protocol (messages `sent0`, `sent4`, `sent2`) and then create a message for the honest user (`sent3`) who then sends message 4 to `C` (`sent1`). This translates to the following attack on the original protocol.

Previously unreported attack on OAuth (RFC version). An attacker performs steps 1-10 of the protocol and delegates access to its account *X* at *P* for a limited period of time to *C*. However, instead of getting redirected to *C*, it induces a victim - having a valid account *V* at *C* – to click a link that contains both request token and verifier. On clicking the link, *V* is transferred to *C*, where it either starts a new login session or continues with an existing session. *C* believes that the valid request token and verifier are for *V*'s account at *P*, while they are actually associated with the attacker's account *X*. If *V* accesses a service at *C* that requires information to be shared with a remote account at *P* (e.g. backing up an address book), *C* inadvertently releases the sensitive information to the attacker.

Fixing the protocol. The protocol can be easily fixed by requiring consumer to include a cookie along with the request token while redirecting user to provider in step 4. It can then check for presence of this cookie when the user returns in step 11. This ensures that steps 4 and 11 are performed by the same user.

When we execute the analyzer again after changing constraints 2 and 3 in the Alloy model to include the cookie, we are not presented with a counter-example even for a scope allowing a protocol sequence of up to 20 messages.

8. CONCLUSIONS

We consider the problem of providing a generic framework for security analysis of web protocols. Belief logics are known to be fast and cost effective tools for this purpose and we feel these techniques could be even better suited for the web domain. However, there are known issues with multiple session attacks and new challenges are introduced due to browser-based communication.

We propose a novel two stage approach where belief logic analysis is followed by automated model finding. The two stages

are linked through a security property which is assumed in the first stage and validated by the second stage. The belief logic we use is our extension of BAN logic for the web domain. It supports principals without identifying keys, secure SSL/TLS channels and simple reasoning about user actions, thus enabling and simplifying analysis of web protocols.

We develop a generic model for web protocol analysis in Alloy. Since scope of this analysis is to check for a particular security property, we make considerable simplifications in both the protocol and intruder models. This significantly reduces complexity of model checking in our approach. We demonstrate our hybrid method through analysis of OAuth, a leading web identity and access management protocol. We identify a previously unreported vulnerability in an approved RFC version which is still being used by several service providers. We also propose a simple fix that service providers can use to overcome the insecurity. Use of lightweight analysis methods makes it practical for our method to be incorporated in design, development of web protocols and standards.

9. REFERENCES

[1] Burrows, M., Abadi, M. and Needham, R. 1990. A Logic of Authentication. *ACM Trans. Comp. Sys*. 8, 1, 18-36.

[2] OASIS SAML Specifications. SAML v2.0, Core. 2005. http://saml.xml.org/saml-specifications

[3] OpenID 2.0 Specifications. 2008. http://openid.net/specs/openid-authentication-2_0.html.

[4] Hammer, E. 2010. *The OAuth 1.0 Protocol. Internet Engineering Task Force, Request for Comments (RFC)*: 5849, http://www.rfc-editor.org/rfc/rfc5849.txt.

[5] Gong, L., Needham, R. and Yahalom, R. 1990. Reasoning about Belief in Cryptographic Protocols. In *Proceedings of IEEE Symposium on Research in Security and Privacy* (1990)

[6] Abadi, M. and Tuttle, M.R. 1991. A semantics for a logic of authentication. In *Proceedings of the ACM Symposium of Principles of Distributed Computing* (1991)

[7] Kessler, V. and Wedel, G. 1994. AUTLOG: An advanced logic of authentication. In *Proceedings of Computer Security Foundation Workshop* VII, 90-99 (1994).

[8] Syverson, P. and van Oorschot, P. 1994. On unifying some cryptographic protocol logics. In *Proceedings of the Symposium on Security and Privacy*, Oakland, CA, 14-28.

[9] Schumann, J. 1997. Automatic Verification of Cryptographic Protocols with SETHEO. In *McCune, W. (ed.) CADE 1997. LNCS*, vol. 1249, 831-836. Springer, Heidelberg.

[10] Craigen, D. and Saaltink, M. 1996. *Using EVES to analyze authentication protocols*. Technical Report TR-96-5508-05, ORA Canada.

[11] Dolev, D. and Yao, A. 1983. On the security of public key protocols. *IEEE Trans. Inform. Theory* IT-29, 198-208.

[12] Meadows, C. 1992. Applying formal methods to the analysis of a key management protocol. *Journal of Computer Security* 1, 5-53.

[13] Lowe, G. 1996. Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR. In Margaria, T., Steffen, B. (eds.) *TACAS 1996. LNCS, 1055, 147-166. Springer, Heidelberg*.

[14] Armando, A., et al. 2005. An Optimized Intruder Model for SAT-based Model-Checking of Security Protocols. *Elec. Notes in Theoret. Comp. Sci.* 125(1) (March 2005)

[15] Software Design Group, MIT, *Alloy analyzer 4*. 2010. http://alloy.mit.edu/

[16] Hammer-Lahav, E. 2009. Explaining the OAuth Session Fixation Attack, http://hueniverse.com/2009/04/explaining-the-oauth-sessionfixation-attack/

[17] Jackson, D. 2002. Alloy: a lightweight object modelling notation. *ACM Trans. on Software Engineering and Methodology* (TOSEM), 11, 2, 256-290.

[18] Basin, D., Modersheim, S. and Vigano, L. 2003. An On-The-Fly Model-Checker for Security Proto-col Analysis. In *Proceedings of 8th ESORICS* 2003, 253-270. LNCS 2808.

[19] Armando, A., Basin, D., Boichut Y., et al. 2005. The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications. In *Proceedings of International Conference on Computer-Aided Verification.*

[20] Javier, F., Fabrega, T., Herzog, J., C. and Guttman, J., D. 1998. Strand spaces: Why a security protocol is correct?. In *Proceedings of IEEE Symposium on Security and Privacy*, 160-171.

[21] Dawn, S., Berezin., S. and Perrig, A. 2001. Athena: a novel approach to efficient automatic security protocol analysis. *Journal of Computer Security*, 9, 47-74.

[22] Akhawe, D., Barth, A., Lam, P., E., Mitchell, J. and Song D. 2010. Towards a Formal Foundation of Web Security. In *Proceedings of 23rd IEEE Computer Security Foundations Symposium (CSF)*, 2010, 290-304.

[23] Kumar, A. 2011. Model Driven Security Analysis of IDaaS Protocols. In *Proceedings of 9th International Conference on Service Oriented Computing.* 312-327.

[24] The OAuth Core 1.0 Specification, 2007. http://oauth.net/core/1.0

[25] Hammer, E., et al., D. 2012. *The OAuth 2.0 Authorization Protocol*, Network Working Group, Internet Draft (work in progress), http://tools.ietf.org/html/draft-ietf-oauth-v2-xx.

[26] Clark, J. and Jacob, J. 1997. *A Survey of Authentication Protocol Literature*: Version 1.0, 17. http://www.eecs.umich.edu/acal/swerve/docs/49-1.pdf.

[27] Armando, A. et al. 2008. Formal analysis of SAML 2.0 web browser single sign-on: breaking the SAML-based single sign-on for google apps. In *Proceedings of 6th ACM workshop on Formal methods in security engineering*.

[28] Kindred, D. and Wing, J. 1996. Fast, Automatic Checking of Security Protocols. In *Proceedings of the USENIX 1996 Workshop on Electronic Commerce*.

[29] Jackson, D. 2012. *Software Abstractions: Logic, Language, and Analysis, Revised Edition*, The MIT Press.

[30] Blanchet, B. 2001. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *Proceedings 14th IEEE Computer Security Foundations Workshop* (CSFW), 82–96.

[31] Cremers, C., J. 2008. Unbounded Verification, Falsification, and Characterization of Security Protocols by Pattern Refinement. In *Proceedings of the 15th ACM conference on Computer and communications security*. 119-128.

Securing Untrusted Code via Compiler-Agnostic Binary Rewriting*

Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, Zhiqiang Lin
Department of Computer Science, The University of Texas at Dallas
800 W. Campbell Rd, Richardson, TX, 75080
{richard.wartell, vishwath.mohan, hamlen, zhiqiang.lin}@utdallas.edu

ABSTRACT

Binary code from untrusted sources remains one of the primary vehicles for malicious software attacks. This paper presents REINS, a new, more general, and lighter-weight binary rewriting and inlining system to tame and secure untrusted binary programs. Unlike traditional monitors, REINS requires no cooperation from code-producers in the form of source code or debugging symbols, requires no client-side support infrastructure (e.g., a virtual machine or hypervisor), and preserves the behavior of even complex, event-driven, x86 native COTS binaries generated by aggressively optimizing compilers. This makes it exceptionally easy to deploy. The safety of programs rewritten by REINS is independently machine-verifiable, allowing rewriting to be deployed as an untrusted third-party service. An implementation of REINS for Microsoft Windows demonstrates that it is effective and practical for a real-world OS and architecture, introducing only about 2.4% runtime overhead to rewritten binaries.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Access controls*; D.3.4 [**Programming Languages**]: Processors—*Code generation*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Validation*

1. INTRODUCTION

Software is often released in binary form. There are numerous distribution channels, such as downloading from the vendor's web site, sharing through a P2P network, or sending via email attachments. All of these channels can introduce and distribute malicious code. Thus, it is very common for end-users to possess known but not fully trusted binary code, or even unknown binaries that they are lured to run. To date, there are two major classes of practical mechanisms to protect users while running such binaries. One is a heavy-weight approach that runs the binary in a contained virtual machine (VM) (e.g., [13, 32, 25]). The other is a lighter-weight approach that runs them in a sandboxing environment with an in-lined reference monitor (IRM) [36, 30, 39].

*This work was supported in part by NSF award #1054629 and AFOSR awards FA9550-08-1-0044 and FA9550-10-1-0088.

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

The VM approach is appealing for several reasons. First, it avoids the problem of statically disassembling CISC binaries. Instead, VMs dynamically translate binary code with the aid of just-in-time binary translation [13, 32, 25, 18]. This allows dynamically computed jump targets to be identified and disassembled on the fly. Second, VMs can intercept API calls and filter them based on a security policy. Third, even if damage occurs, it can potentially be contained within the VM. Therefore, VM approach has been widely used in securing software and analyzing malicious code.

However, production-level VMs can be extremely large relative to the untrusted processes they guard, introducing significant computational overhead when they are applied to enforce fine-grained policies. Their high complexity also makes them difficult to formally verify; a single bug in the VM implementation leaves users vulnerable to attack. Meanwhile, there is an air-gap if the binary needs to access host files, and VM services must also bridge the semantic-gap [7]. While lighter-weight VM alternatives, such as program shepherding [18], lessen some of these drawbacks, they still remain larger and slower than IRMs.

On the other hand, a large body of past research (e.g., SFI [36], PittSFIeld [20], CFI [1], XFI [10], NaCl [39]) has recognized the many advantages of client-side, static, binary-rewriting for securing untrusted, mobile, native code applications. Binary-rewriting boasts great deployment flexibility since it can be implemented separately from the code-producer (e.g., by the code-consumer or a third party), and the rewritten code can be safely and transparently executed on machines with no specialized security hardware, software, or VMs. Moreover, it offers superior performance to many VM technologies since it statically in-lines a light-weight VM logic directly into untrusted code, avoiding overheads associated with context-switching and dynamic code generation. Finally, safety of rewritten binaries can be machine-verified automatically (in the fashion of proof-carrying-code [22]), allowing rewriting to be performed by an untrusted third party.

Unfortunately, all past approaches to rewriting native binary code require some form of cooperation from code-producers. For example, Google's Native Client (NaCl) [39] requires a special compiler to modify the client programs at the source level and use NaCl's trusted libraries. Likewise, Microsoft's CFI [1] and XFI [10] requires code-producers to supply a *program database* (PDB) file (essentially a debug symbol table) with their released binaries. Earlier works such as PittSFIeld [20] and SASI [11] require code-producers to provide `gcc`-produced assembly code. Code that does not satisfy these requirements cannot be rewritten and is therefore conservatively rejected by these systems. These restrictions have prevented binary-rewriting from being applied to the vast majority of native binaries because most code-producers do not provide such support and are unlikely to do so in the near future.

Therefore, in this paper we present the first, purely static, CISC

native code rewriting and in-lining system (REINS) that requires no cooperation from code-producers (i.e., is compiler-agnostic). Unlike past work, REINS can automatically rewrite large-scale, COTS, Windows applications yielded by arbitrary compilers, even with no access to source-level information (e.g., PDB files or debug symbol stores). It transparently supports a large category of production-level applications unsupportable by past efforts, including those that include event-driven OS-callbacks, dynamic linking, exceptions, multithreading, computed jumps, and mixtures of trusted and untrusted modules. In past efforts we have successfully used REINS' core rewriting engine to implement basic block randomization for over 100 Windows and Linux applications without source code [37].

Realizing REINS for COTS x86 binary in Windows platform raises many challenges, including semantic preservation of dynamically computed jumps, code interleaved with data, function callbacks, and imperfect disassembly. We address these challenges through the design and implementation of a suite of novel techniques, including conservative disassembly and indirect jump target identification. Central to our approach is a binary transformation strategy that expects and tolerates many forms of disassembly errors by conservatively treating every byte in target code sections as both code and static data. This obviates the need for perfect disassemblies, which are seldom realizable in practice without source code.

To tame and secure unsafe logic inside the binary code, REINS automatically transforms binaries to redirect system API calls through a trusted *policy-enforcement library*. The library thereby mediates all security-relevant API calls and their arguments before (and after) they are serviced, and uses this information to enforce safety policies over histories of these security-relevant events. Indirect control flow transfers (e.g., `call/jmp/ret`) are protected by in-lined guard code that ensures that they target safe code addresses when executed. In addition, a small, trusted verifier shifts the significant complexity of the rewriting system out of the trusted computing base (TCB) by independently certifying that rewritten binaries cannot circumvent the in-lined monitor. Thus, binaries that pass verification are guaranteed to be safe to execute. While reflective code can change its behavior in response to rewriting, verification ensures that such changes cannot effect policy-violations.

In summary, REINS makes the following contributions:

- We present the first *compiler-agnostic, machine-certifying, x86 rewriting* algorithm that supports real-world COTS binaries without any appeal to source code or debug symbols. To the best of our knowledge, all past static binary rewriting techniques require source-level or debugging information to support many COTS binary features.
- We design a set of novel techniques to support binary families for which fully correct automated disassembly is provably undecidable, including those that contain computed jumps, dynamic linking, static data interleaved with code, and untrusted callback functions invoked by the OS.
- We have implemented REINS as a proof-of-concept prototype, and tested it on a number of binaries including malware code. Our empirical evaluation shows that our system successfully preserves the behavior of non-malicious, real-world Windows applications, introducing runtime overheads of about 2.4%.

2. BACKGROUND AND OVERVIEW

2.1 Background

Assumptions. The goal of our system is to tame and secure malicious code in untrusted binaries through static binary rewriting. Since a majority of malware threats currently target Windows x86 platforms, we assume the binary code is running in Microsoft Windows OS with x86 architecture. Protecting Linux binary code is outside the scope of this paper. (In fact, rewriting Windows binary code is much more challenging than for Linux due to the much greater diversity of Windows-targeting compilers.)

Our goal is to design a compiler-agnostic static binary rewriting technique, so we do not impose any constraints on the code-producer; it could be any Windows platform compiler, or even hand-written machine code. Debug information (e.g., PDB) is assumed to be unavailable. Like all past native code IRM systems, our fully static approach rejects attempts at self-modification; untrusted code may only implement runtime-generated code through standard system API calls, such as dynamic link library (DLL) loading. Code-injection attacks are therefore thwarted because the monitor ensures that any injected code is unreachable.

In addition, our goal is not to protect untrusted code from harming itself. Rather, we prevent modules that may have been compromised (e.g., by a buffer overflow) from abusing the system API to damage the file system or network, and from corrupting trusted modules (e.g., system libraries) that may share the untrusted module's address space. This confines any damage to the untrusted module.

Threat model. Attackers in our model submit arbitrary x86 binary code for execution on victim systems. Neither attackers nor defenders are assumed to have kernel-level (ring 0) privileges. Attacker-supplied code runs with user-level privileges, and must therefore leverage kernel-supplied services to perform malicious actions, such as corrupting the file system or accessing the network to divulge confidential data. The defender's ability to thwart these attacks stems from his ability to modify attacker-supplied code before it is executed. His goal is therefore to reliably monitor and restrict access to security-relevant kernel services without the aid of kernel modifications or application source code, and without impairing the functionality of non-malicious code.

Attacks. The central challenge for any protection mechanism that constrains untrusted native code is the problem of taming computed jumps, which dynamically compute control-flow destinations at runtime and execute them. Attackers who manage to corrupt these computations or the data underlying them can hijack the control-flow, potentially executing arbitrary code.

While computed jumps may seem rare to those accustomed to source-level programming, they actually pervade almost all binary programs compiled from all source languages. Computed jumps typically include returns (whose destinations are drawn from stack data), method calls (which use method dispatch tables), library calls (which use import address tables), multi-way branches (e.g., switch-case), and optimizations that cache code addresses to registers.

Deciding whether any of these jumps might target an unsafe location at runtime requires statically inferring the program register and memory state at arbitrary code points, which is a well known undecidable problem. Moreover, since x86 instructions are unaligned (i.e., any byte can be the start of an instruction), computed jumps make it impossible to reliably identify all instructions in untrusted binary code; disassemblers must heuristically guess the addresses of many instruction sequences to generate a complete disassembly. Untrusted binaries (e.g., malicious code) are often specifically crafted to defeat these heuristics, thereby concealing malicious instruction sequences from analysis tools.

2.2 System Overview

Given an untrusted binary, REINS automatically transforms it so that (1) all access to system (and library) APIs are mediated by our policy enforcement library, and (2) all inter-module control-flow

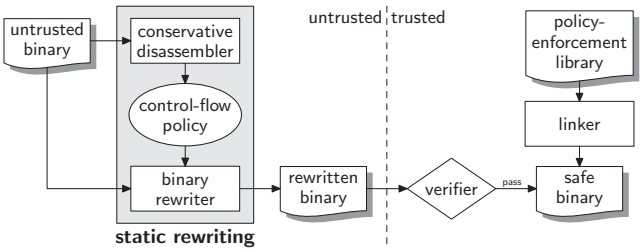


Figure 1: REINS **architecture**

transfers are restricted to published entry points of known libraries, preventing execution of attacker-injected or misaligned code.

REINS' rewriter first generates a conservative disassembly of the untrusted binary that identifies all safe, non-branching flows (some of which might not actually be reachable) but not unsafe ones. The resulting disassembly encodes a control-flow policy: instructions not appearing in the disassembly are prohibited as computed jump targets. Generating even this conservative disassembly of arbitrary x86 COTS binaries is challenging because COTS code is typically aggressively interleaved with data, and contains significant portions that are only reachable via computed jumps. To help overcome some of these challenges, our rewriter is implemented as an IDAPython [9] program that leverages the considerable analysis power of the Hex-rays IDA Pro commercial disassembler to identify function entrypoints and distinguish code from data in complex x86 binaries. While IDA Pro is powerful, it is not perfect; it suffers numerous significant disassembly errors for almost all production-level Windows binaries. Thus, our rewriting algorithm's tolerance of disassembly errors is critical for success.

Our system architecture is illustrated in Fig. 1. Untrusted binaries are first analyzed and transformed into safe binaries by a binary rewriter, which enforces control-flow safety and mediates all API calls. A separate verifier certifies that the rewritten binaries are policy-adherent. Malicious binaries that defeat the rewriter's analysis might result in rewritten binaries that fail verification or that fail to execute properly, but never in policy violations.

3. DETAILED DESIGN

3.1 Rewriting Control-flow Transfers

Control Flow Safety. Our binary rewriting algorithm uses SFI [36] to constrain control-flows of untrusted code. It is based on an SFI approach pioneered by PittSFIeld [20], which partitions instruction sequences into c-byte *chunks*. Chunk-spanning instructions and targets of jumps are moved to chunk boundaries by padding the instruction stream with `nop` (no-operation) instructions. This serves three purposes:

- When c is a power of 2, computed jumps can be efficiently confined to chunk boundaries by guarding them with an instruction that dynamically clears the low-order bits of the jump target.
- Co-locating guards and the instructions they guard within the same chunk prevents circumvention of the guard by a computed jump. A chunk size of $c = 16$ suffices to contain each guarded sequence in our system.
- Aligning all code to c-byte boundaries allows a simple, fall-through disassembler to reliably discover all reachable instructions in rewritten programs, and verify that all computed jumps are suitably guarded.

To allow trusted, unrewritten system libraries to safely coexist in the same address space as chunk-aligned, rewritten binaries, we logically divide the virtual address space of each untrusted process into *low memory* and *high memory*. Low memory addresses range from 0 to $d-1$ and may contain rewritten code and non-executable data. Higher memory addresses may contain code sections of trusted libraries and arbitrary data sections (but not untrusted code).

Partition point d is chosen to be a power of 2 so that a single guard instruction suffices to confine untrusted computed jumps and other indirect control flow transfers to chunk boundaries in low memory. For example, a jump that targets the address currently stored in the `eax` register can be guarded by:

```
and eax, (d − c)
jmp eax
```

This clears both the high-order and low-order bits of the target address before jumping, preventing an untrusted module from jumping directly to a system accessor function or to a non-chunk boundary in its own code, respectively. The partitioning of virtual addresses into low and high memory is feasible because rewritten code sections are generated by the rewriter and can therefore be positioned in low memory, while trusted libraries are relocatable through *rebasing* and can therefore be moved to high memory when necessary.

Preserving Good Flows. The above suffices to enforce control-flow safety, but it does not preserve the behavior of most code containing computed jumps. This is a major deficiency of many early SFI works, most of which can only be successfully applied to relatively small, `gcc`-compiled programs that do not contain such jumps. More recent SFI works have only been able to overcome this problem with the aid of source-level debug information.

Our source-free solution capitalizes on the fact that although disassemblers cannot generally identify all jumps in arbitrary binary code, modern commercial disassemblers can heuristically identify a *superset* of all the indirect jump *targets* (though not the jumps that target them) in most binary code. This is enough information to implement a light-weight, binary lookup table that the IRM can consult at runtime to dynamically detect and correct computed jump targets before they are used. Our lookup table overwrites each old target with a tagged pointer to its new location in the rewritten code. This solves the computed jump preservation problem without the aid of source code.

Since the disassembler identifies a superset of targets and not an exact set, the lookup table implementation must be carefully designed to tolerate false positives. Misidentification of a code point as a jump target is therefore relatively harmless to REINS; each such misidentification merely increases the size of rewritten code by a few bytes due to alignment. A false negative (i.e., failure to identify one or more targets) is more serious and may lead to rewritten code that does not execute properly, but the verifier ensures that it cannot lead to a policy violation. Thus, both forms of error are tolerated.

Another major design issue is the need to arrange the lookup table so that IRM code that uses it remains exceptionally small and efficient. This is critical for achieving low overhead, since computed jumps are extremely common in real-world binaries. Our solution implements most lookups with just two non-branching instructions (a compare instruction and a conditional move), shown atop the first row of Table 1. This efficient implementation is achieved by tagging each lookup table entry with a leading byte that never appears as the first byte of valid code. We use a tag byte of `0xF4`, which encodes an x86 `hlt` instruction that is illegal in protected mode. The compare instruction uses this byte to quickly distinguish stale pointers that point into the lookup table from those that already point to code. The conditional move then corrects the stale ones. This succinct realization of semantics-preserving computed jump guards is the key to REINS' exceptionally low overhead.

Retaining the old code section as a data section has the additional advantage of retaining any static data that may be interleaved in the

Table 1: Summary of x86 code transformations

Description	Original code	Rewritten code

Computed jumps with register operands	`call/jmp` r	`cmp byte ptr [`r`], 0xF4`
		`cmovz` r`, [`r`+1]`
		`and` r`, (`$d-c$`)`
		`call/jmp` r

Computed jumps with memory operands	`call/jmp [`m`]`	`mov eax, [`m`]`
		`cmp byte ptr [eax], 0xF4`
		`cmovz eax, [eax+1]`
		`and eax, (`$d-c$`)`
		`call/jmp eax`

Returns	`ret (`n`)`	`and [esp], (`$d-c$`)` `ret (`n`)`
IAT loads	`mov` rm`, [IAT:`n`]`	`mov` rm`, offset tramp_`n

Tail-calls to high memory	`jmp [IAT:`n`]`	`tramp_`n`:`
		`and [esp], (`$d-c$`)`
		`jmp [IAT:`n`]`

code. This data can therefore be read by the rewritten executable at its original addresses, avoiding many difficult data preservation problems that hamper other SFI systems. The tradeoff is an increased size of rewritten programs, which tend to be around twice the size of the original. However, this does not necessarily lead to an equivalent increase in runtime process sizes. Our experiences with real x86 executables indicates that dynamic data sizes tend to eclipse static code sizes in memory-intensive processes. Thus, in most cases rewritten process sizes incur only a fraction of the size increase experienced by the disk images whence they were loaded.

When the original computed jump employs a memory operand instead of a register, as shown in row 2 of Table 1, the rewritten code requires a scratch register. Table 1 uses `eax`, which is caller-save by convention and is not used to pass arguments by any calling convention supported by any mainstream x86 compiler [12].[1]

A particularly common form of computed jump deserves special note. Return instructions (`ret`) jump to the address stored atop the stack (and optionally pop n additional bytes from the stack afterward). These are guarded by the instruction given in row 3 of Table 1, which masks the return address atop the stack to a low memory chunk boundary. Call instructions are moved to the ends of chunks so that the return addresses they push onto the stack are aligned to the start of the following chunk. Thus, the return guards have no effect upon return addresses pushed by properly rewritten call instructions, but they block jumps to corrupted return addresses that point to illegal destinations, such as the stack. This makes all attacker-injected code unreachable.

Preserving API Calls. To allow untrusted code to safely access trusted library functions in high memory, the rewriter permits one form of computed jump to remain unguarded: Computed jumps whose operands directly reference the *import address table* (IAT) are retained. Such jumps usually have the following form:

`call [IAT:`n`]`

where `IAT` is the section of the executable reserved for the IAT and n is an offset that identifies the IAT entry. These jumps are safe since the entrypoint to the APIs is hooked by REINS to ensure that they always target policy-compliant addresses at runtime.

Not all uses of the IAT have this simple form, however. Most x86-targeting compilers also generate optimized code that caches IAT entries to registers, and uses the registers as jump targets. To safely accommodate such calls, the rewriter identifies and modifies all instructions that use IAT entries as data. An example of such an instruction is given in row 4 of Table 1. For each such instruction, the rewriter replaces the IAT memory operand with the address of a callee-specific *trampoline chunk* (in row 5) introduced to the rewritten code section (if it doesn't already exist). The trampoline chunk safely jumps to the trusted callee using a direct IAT reference. Thus, any use of the replacement pointer as a jump target results in a jump to the trampoline, which invokes the desired function.

Dynamic linking and callbacks are both supported via a similar form of trampolining detailed in the technical report [16].

[1]To support binaries that depend on preserving `eax` across computed jumps, the table's sequence can be extended with two instructions that save and restore `eax`. We did not encounter any programs that require this, so our experiments use the table's shorter sequence.

```
Original:
.text:00499345  8B 35 FC B5 4D 00     mov esi, [4DB5FCh] ;IAT:MBTWC
...
.text:00499366  FF D6                 call esi

Rewritten:
.tnew:0059DBF0  BE 90 12 5D 00        mov esi, offset loc_5D1290
...
.tnew:0059DC15  80 3E F4              cmp byte ptr [esi], F4h
.tnew:0059DC18  0F 44 76 01           cmovz esi, [esi+1]
.tnew:0059DC1C  90 90 90 90           nop (×4)
.tnew:0059DC20  81 E6 F0 FF FF 0F     and esi, 0FFFFFF0h
.tnew:0059DC26  90 (×8)               nop (×8)
.tnew:0059DC2E  FF D6                 call esi
...
.tnew:005D1290  81 24 24 F0 FF FF 0F  and dword ptr [esp], 0FFFFFF0h
.tnew:005D1297  FF 25 FC B5 4D 00     jmp [4DB5FCh] ;IAT:MBTWC
```

Figure 2: Rewriting a register-indirect system call

```
Original:
.text:00408495  FF 24 85 CC 8A 40 00  jmp ds:off_408ACC[eax*4]
...
.text:00408881  3D 8C 8A 4D 00 00     cmp byte_4D8A8C, 0
.text:00408888  74 13                 jz short loc_40889D
.text:0040888A  84 C9                 test cl, cl
.text:0040888C  74 0F                 jz short loc_40889D
...
.text:00408ACC  81 88 40 00           dd offset loc_408881
.text:00408AD0  ...                   (other code pointers)

Rewritten:
.text:00408881  F4 60 3A 4F 00        db F4, loc_4F3A60
                — — — — — — — — — — — — — — — —
.tnew:004F33B4  8B 04 85 CC 8A 40 00  mov eax, ds:dword_408ACC[eax*4]
.tnew:004F33BB  80 38 F4              cmp byte ptr [eax], F4h
.tnew:004F33BE  90 90                 nop (×2)
.tnew:004F33C0  0F 44 40 01           cmovz eax, [eax+1]
.tnew:004F33C4  25 F0 FF FF 0F        and eax, 0FFFFFF0h
.tnew:004F33C9  FF E0                 jmp eax
...
.tnew:004F3A60  3D 8C 8A 4D 00        cmp byte_4D8A8C, 0
.tnew:004F3A67  74 27                 jz short loc_4F3A90
.tnew:004F3A69  84 C9                 test cl, cl
.tnew:004F3A6B  74 22                 jz short loc_4F3A90
```

Figure 3: Rewriting code that uses a jump table

3.2 Examples

To illustrate the rewriting algorithm, Figs. 2 and 3 demonstrate the transformation process for two representative assembly codes.

Figure 2 implements a register-indirect call to a system API function (MBTWC). The first instruction of the original code loads an IAT entry into the `esi` register, which is later used as the target of the call. REINS replaces this address with the address of the in-lined trampoline code at the bottom of the figure, which performs a safe jump to the same destination. The call instruction is replaced with the guarded call sequence shown in lines 2–7 of the rewritten binary. The compare (`cmp`) and conditional move (`cmovz`) implement the table-lookup, and the masking instruction (`and`) aligns the destination to a chunk boundary. This makes the ensuing call provably safe to execute.

Figure 3 shows a computed jump with a memory operand that indexes a jump table. The rewritten code first loads the destination address into a scratch register (`eax`) in accordance with row 2 of Table 1. It then implements the same lookup and masking guards as in Fig. 2. This time the lookup has a significant effect—it discovers at runtime that the address drawn from the lookup table must be repointed to a new address. This preserves the behavior of the binary after rewriting despite the failure of the disassembler to discover and identify the jump table at rewrite-time.

3.3 Memory Safety

To prevent untrusted binaries from dynamically modifying code sections or executing data sections as code, untrusted processes are executed with DEP enabled. DEP-supporting operating systems allow memory pages to be marked non-executable (NX). Attempts to execute code in NX pages result in runtime access violations. The binary rewriter sets the NX bit on the pages of all low memory sections other than rewritten code sections to prevent them from being executed as code. Thus, attacker-injected shell code in the stack or other data memory regions cannot be executed.

User processes on Windows systems can set or unset the NX bit on memory pages within their own address spaces, but this can only be accomplished via a small collection of system API functions—e.g., `VirtualProtect` and `VirtualAlloc`. The rewriter replaces the IAT entries of these functions with trusted wrapper functions that silently set the NX bit on all pages in low memory other than rewritten code pages. The wrappers do not require any elevated privileges; they simply access the real system API directly with modified arguments.

The real system functions are accessible to trusted libraries (but not untrusted libraries) because they have separate IATs that are not subjected to our IAT hooking. Trusted libraries can therefore use them to protect their local heap and stack pages from untrusted code that executes in the same address space. Our API hooks prevent rewritten code from directly accessing the page protection bits to reverse these effects. This prevents the rewritten code from gaining unauthorized access to trusted memory.

Our memory safety enforcement strategy conservatively rejects untrusted, self-modifying code. Such code is a mainstay of certain application domains, such as JIT-compilers. For these domains we consider alternative technologies, such as certifying compilers and certified, bytecode-level IRMs, to be a more appropriate means of protection. Self-modifying code is increasingly rare in other domains, such as application installers, because it is incompatible with DEP, incurs a high performance penalty, and tends to trigger conservative rejection by antivirus products. No SFI system to our knowledge supports arbitrary self-modifying code.

3.4 Verification

The disassembler, rewriter, and lookup table logic all remain completely untrusted by our architecture. Instead, a small, independent verifier certifies that rewritten programs cannot circumvent the IAT and are therefore policy-adherent. The verifier does not prove that the rewriting process is behavior-preserving. This reduced obligation greatly simplifies the verifier relative to the rewriter, resulting in a small TCB.

The verification algorithm performs a simple fall-through disassembly of each executable section in the untrusted binary and checks the following purely syntactic properties:

- All executable sections reside in low memory.
- All exported symbols (including the program entrypoint) target low memory chunk boundaries.
- No disassembled instruction spans a chunk boundary.
- Static branches target low memory chunk boundaries.
- All computed jump instructions that do not reference the IAT are immediately preceded by the appropriate `and`-masking instruction from Table 1 in the same chunk.
- Computed jumps that read the IAT access a properly aligned IAT entry, and are preceded by an `and`-mask of the return address. (Call instructions must *end* on a chunk boundary rather than requiring a mask, since they push their own return addresses.)
- There are no trap instructions (e.g., `int` or `syscall`).

These properties ensure that any unaligned instruction sequences concealed within untrusted, executable sections are not reachable at runtime. This allows the verifier to limit its attention to a fall-through disassembly of executable sections, avoiding any reliance upon the incomplete code-discovery heuristics needed to produce full disassemblies of arbitrary (non-chunk-aligned) binaries.

4. IMPLEMENTATION

We have developed an implementation of REINS for the 32-bit version of Microsoft Windows XP/Vista/7/8. The implementation consists of four components: (1) a rewriter, (2) a verifier, (3) an API hooking utility, and (4) an intermediary library that handles dynamic linking and callbacks. Rather than using a single, static API hooking utility, we implemented an automated *monitor synthesizer* that generates API hooks and wrappers from a declarative policy specification. This is discussed in §5.3. None of the components require elevated privileges. While the implementation is Windows-specific, we believe the general approach is applicable to any modern OS that supports DEP technology.

The rewriter transforms Windows Portable Executable (PE) files in accordance with the algorithm in §3. Its implementation consists of about 1,300 lines of IDA Python scripting code that executes atop the Hex-rays IDA Pro 6.1 disassembler. One of IDA Pro's primary uses is as a malware reverse engineering and de-obfuscating tool, and it boasts many powerful code analyses that heuristically recover program structural information without assistance from a code-producer. These analyses are leveraged by our system to automatically distinguish code from data and identify function entrypoints to facilitate rewriting.

In contrast to the significant complexity of the rewriting infrastructure, the verifier's implementation consists of 1,500 lines of 80-column OCaml code that uses no external libraries or utilities (other than the built-in OCaml standard libraries). Of these 1,500 lines, approximately 1,000 are devoted to x86 instruction decoding, 300 to PE binary parsing, and 200 to the actual verification algorithm in §3.4. The decoder handles the entire x86 instruction set, including floating point, MMX, and all SSE extensions documented in the Intel and AMD manuals. This is necessary for practical testing since production-level binaries frequently contain at least some exotic instructions. No code is shared between the verifier and rewriter.

The intermediary library consists of approximately 500 lines of C and hand-written, in-lined assembly code that facilitates callbacks and dynamic linking. An additional 150-line configuration file itemizes all trusted callback registration functions exported by Windows libraries used by the test programs. We supported all callback registration functions exported by `comdlg32`, `gdi32`, `kernel32`, `msvcrt`, and `user32`. Information about exports from these libraries was obtained by examining the C header files for each library and identifying function pointer types in exported function prototypes.

Our API hooking utility replaces the IAT entries of all monitored system functions imported by rewritten PE files with the addresses of trusted monitor functions. It also adds the intermediary library to the PE's list of imported modules. To avoid expanding the size of the PE header (which could shift the positions of the binary sections that follow it), our utility simply changes the library name `kernel32.dll` in the import section to the name of our intermediary library. This causes the system loader to draw all IAT entries previously imported from `kernel32.dll` from the intermediary library instead. The intermediary library exports all `kernel32` symbols as forwards to the real `kernel32`, except for security-relevant functions, which it exports as local replacements. Our intermediary library thus doubles as the policy enforcement library.

Table 2: Experimental results: SPEC benchmarks

Binary Program	Size Increase			Rewriting Time (s)	Verification Time (ms)
	File (%)	Code (%)	Process (%)		
gzip	103	31	0	12.5	142
vpr	94	26	22	14.4	168
mcf	108	32	2	10.5	84
parser	108	34	1	17.4	94
gap	118	42	0	31.2	245
bzip2	102	29	0	10.8	91
twolf	99	24	27	25.3	245
mesa	104	20	6	42.4	554
art	108	33	14	12.4	145
equake	103	27	1	12.3	165
median	+103.5%	+30.0%	+1.5%	13.45s	155ms

Table 3: Experimental results: Applications and malware

Binary Program	Size Increase			Rewriting Time (s)	Verification Time (ms)
	File (%)	Code (%)	Process (%)		
notepad	60	31	20	1.5	18
Eureka	32	53	15	17.9	225
DOSBox	112	38	0	137.1	2394
PhotoView	87	57	4	3.5	49
BezRender	128	55	3	4.1	55
gcc	100	37	15	3.0	36
g++	100	41	16	3.0	37
jar	101	34	12	2.4	27
objcopy	122	49	23	26.9	354
size	103	50	116	16.3	20
strings	122	50	42	21.5	283
as	99	49	2	30.4	397
ar	121	50	4	21.8	285
whetstone	88	21	54	0.6	6
linpack	57	19)	31	0.6	6
pi_ccs5	125	28	1	5.8	66
md5	25	48	149	0.6	5
median	100%	41%	15%	4.1s	49ms
Virut.a		(rejected)		–	–
Hidrag.a		(rejected)		–	–
Vesic.a	75	34	108	0.3	194
Sinn.1396	37	115	93	0.2	75
Spreder.a	14	66	17	3.0	72
median	37%	66%	93%	0.3s	75ms

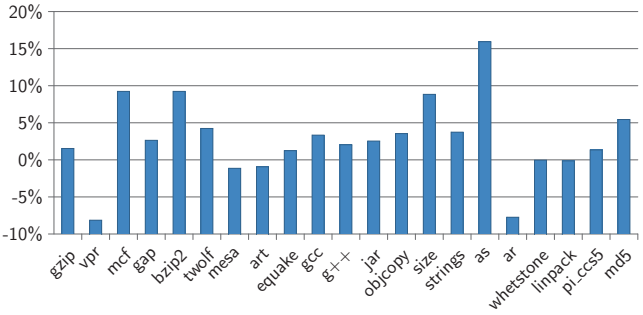


Figure 4: Runtime overhead due to rewriting

5. EVALUATION

5.1 Rewriting Effectiveness

We tested REINS with a set of binary programs listed in Tables 2 and 3. Table 2 lists results for some of the benchmarks from the SPEC 2000 benchmark suite. Table 3 lists results for some other applications, including GUI programs that include event- and callback-driven code, and malware samples that require enforcement of higher-level security policies to prevent malicious behavior. In both tables, columns 2–3 report the percentage increase of the file size, code segment, and process size, respectively; and columns 5–6 report the time taken for rewriting and verification, respectively. All experiments were performed on a 3.4GHz quad-processor AMD Phenom II X4 965 with 4GB of memory running Windows XP Professional and MinGW 5.1.6.

File sizes double on average after rewriting for benign applications, while malware shows a smaller increase of about 40%. Code segment sizes increase by a bit less than half for benign applications, and a bit more than half for malware. Process sizes typically increase by about 15% for benign applications, but almost 90% for malware. The rewriting speed is about 32s per megabyte of code, while verification is much faster—taking only about 0.4s per megabyte of code on average.

5.2 Performance Overhead

We also measured the performance of the non-interactive programs in Tables 2 and 3. The runtime overheads of the rewritten programs as a percentage of the runtimes of the originals is presented in Fig. 4. The median overhead is 2.4%, and the maximum is approximately 15%. As with other similar works [1, 13], the runtimes of a few programs decrease after rewriting. This effect is primarily due to improved instruction alignment introduced by the rewriting algorithm, which improves the effectiveness of instruction look-ahead and decoding pipelining optimizations implemented by modern processors. While the net effect is marginal, it is enough to offset the overhead introduced by the rest of the protection system in these cases, resulting in safe binaries whose runtimes are as fast as or faster than the originals.

The experiments reported in Tables 2 and 3 enforced only the core access control policies required to prevent control-flow and memory safety violations. Case studies that showcase the framework's capacity to enforce more useful policies are described in §5.4.

5.3 Policy Enforcement Library Synthesis

To quickly and easily demonstrate the framework's effectiveness for enforcing a wide class of safety policies, we developed a monitor synthesizer that automatically synthesizes the policy enforcement portion of the intermediary library from a declarative policy specification. Policy specifications consist of: (1) the module names and signatures of all security-relevant API functions to be monitored, (2) a description of the runtime argument values that, when passed to these API functions, constitute a security-relevant *event*, and (3) a regular expression over this alphabet of events whose prefix-closure is the language of permissible *traces* (i.e., event sequences).

To illustrate, Fig. 5 shows a sample policy. Lines 1–5 are signatures of two API functions exported by Windows system libraries: one for connecting to the network and one for creating files. Lines 7–8 identify network-connects as security-relevant when the outgoing port number is 25 (i.e., an SMTP email connection) and the return value is 0 (i.e., the operation was successful), and file-creations as security-relevant when the filename's extension is `.exe`. Underscores denote arguments whose values are not security-relevant. Finally, line 10 defines traces that include at most one kind of event (but not both) as permissible. Here, $\star$ denotes finite or infinite repetition and + denotes regular alternation.

Currently our synthesizer implementation supports dynamic value tests that include string wildcard matching, integer equality and inequality tests, and conjunctions of these tests on fields within a structure. From this specification, the monitor synthesizer generates the C source code of a policy enforcement library that uses IAT hooking to reroute calls to `connect` and `CreateFileW` through trusted guard functions. The guard functions implement the desired

```
1  function conn = ws2_32::connect(
2    SOCKET, struct sockaddr_in *, int) -> int;
3  function cfile = kernel32::CreateFileW(
4    LPCWSTR, DWORD, DWORD, LPSECURITY_ATTRIBUTES,
5    DWORD, DWORD, HANDLE) -> HANDLE WINAPI;

7  event e1 = conn(_, {sin_port=25}, _) -> 0;
8  event e2 = cfile("*.exe", _, _, _, _, _, _) -> _;

10 policy = e1* + e2*;
```

Figure 5: A policy that prohibits applications from both sending emails and creating `.exe` files

```
1  function cfile = kernel32::CreateFileW(
2    LPCWSTR, DWORD, DWORD, LPSECURITY_ATTRIBUTES,
3    DWORD, DWORD, HANDLE) -> HANDLE WINAPI;
4  function exec = kernel32::WinExec(LPCSTR, UINT)
5    -> UINT WINAPI;

7  event e1 = cfile("*.exe", _, _, _, _, _, _) -> _;
8  event e2 = cfile("*.msi", _, _, _, _, _, _) -> _;
9  event e3 = cfile("*.bat", _, _, _, _, _, _) -> _;
10 event e4 = exec("explorer", _) -> _;

12 policy = ;
```

Figure 6: Eureka email policy

policy as a determinized *security automaton* [30]—a finite state automaton that accepts the prefix-closure of the policy language in line 10. If the untrusted code attempts to exhibit a prohibited trace, the monitor rejects by halting the process.

5.4 Case Studies

5.4.1 An Email Client

As a more in-depth case-study, we used the rewriting system and monitor synthesizer to enforce two policies on the *Eureka* 2.2q email client. Eureka is a fully featured, commercial POP client for 32-bit Windows that features a graphical user interface, email filtering, and support for launching executable attachments as separate processes. It is 1.61MB in size and includes all of the binary features discussed in earlier sections, including Windows event callbacks and dynamic linking. It statically links to eight trusted system libraries.

Without manual assistance, IDA automatically recovers enough structural information from the Eureka binary to facilitate the full binary rewriting algorithm presented in §3. Rewriting requires 18s and automated verification of the rewritten binary requires 0.2s.

After rewriting, we synthesized an intermediary library that enforces the access control policy given in Fig. 6, which prohibits creation of files whose filename extensions are `.exe`, `.msi`, or `.bat`, and which prevents the application from launching Windows Explorer as an external process. (The empty policy expression in line 12 prohibits all events defined in the specification.) We also enforced the policy in Fig. 5, but with a policy expression that limits clients to at most 100 outgoing SMTP connections per run. Such a policy might be used to protect against malware infections that hijack email applications for propagation and spamming.

After rewriting, we systematically tested all program features and could not detect any performance degradation or changes to any policy-permitted behaviors. All program features unrelated to the policy remain functional. However, saving or launching an email attachment with any of the policy-prohibited filename extensions causes immediate termination of the program by the monitor. Likewise, using any program operation that attempts to open an attachment using Windows Explorer, or sending more than 100 email messages, terminates the process. The rewritten binary therefore correctly enforces the desired policy without impairing any of the application's other features.

5.4.2 An Emulator

DOSBox is a large DOS emulator with over 16 million downloads on sourceforge. Though its source code is available, it was not used during the experiment. The precompiled binary is 3.6MB, and like Eureka, includes all the difficult binary features discussed earlier.

We enforced several policies that prohibit access to portions of the file system based on filename string and access mode. We then used the rewritten emulator to install and use several DOS applications, including the games Street Fighter 2 and Capture the Flag. Installation of these applications requires considerable processing time, and is the basis for the timing statistics reported in Table 3. As in the previous experiment, no performance degradation or behavioral changes are observable in the rewritten application, except that policy-violating behaviors are correctly prohibited.

5.4.3 Malware

To analyze the framework's treatment of real-world malware, we tested REINS on five malware samples obtained from a public malware research repository: *Virut.a*, *Hidrag.a*, *Vesic.a*, *Sinn.1396*, and *Spreder.a*. While these malware variants are well-known and therefore preventable by conventional signature-matching antivirus defenses, the results indicate how our system reacts to binaries intentionally crafted to defeat disassembly tools and other static analyses. Each is statically or dynamically rejected by the protection system at various different stages, detailed below.

Virut and *Hidrag* are both rejected at rewriting time when the rewriter encounters misaligned static branches that target the interior of another instruction. While supporting instruction aliasing due to misaligned *computed* jumps is useful for tolerating disassembly errors, misaligned *static* jumps only appear in obfuscated malware to our knowledge, and are therefore conservatively rejected.

Vesic and *Sinn* are Win32 viruses that propagate by appending themselves to executable files on the C: volume. They do not use packing or obfuscation, making them good candidates for testing our framework's ability to detect malicious behavior rather than just suspicious binary syntax. With a fully permissive policy, our framework successfully rewrites and verifies both malware binaries; running the rewritten binaries preserves their original (malicious) behaviors. However, enforcing the policy in Fig. 6 results in premature termination of infected processes when they attempt to propagate by writing to executable files. We also successfully enforced a second policy that prohibits the creation of system registry keys, which *Vesic* uses to insert itself into the boot process of the system. These effectively protect the infected system before any damage results.

Spreder has a slightly different propagation strategy that searches for executable files in the shared directory of the Kazaa file-sharing peer-to-peer client. We successfully enforced a policy that prohibits use of the `FindFirstFileA` system API function to search for executable files in this location. This results in immediate termination of infected processes.

6. DISCUSSION

In this section we first discuss the security benefits REINS provides, and then discuss the binary code conventions that are prerequisites for behavior-preservation under our binary rewriting scheme, as well as the reliability of our disassembly. The limitations of our approach are highlighted during the course of the discussion.

6.1 Control-flow Policies

As we have demonstrated, REINS can rewrite many complex legacy binaries, enforcing coarse-grained control-flow safety and preserving safe computed jumps without source code. However, REINS does not enforce the finer-grained control-flow integrity properties of CFI [1]. CFI uses source code or PDB files to build a control-flow graph that serves as the integrity policy to enforce. This connection to source code is foundational to CFI because any fine-grained definition of "good" control-flows invariably depends on the semantics of the source code that the untrusted binary code is intended to reflect. Without source code, there is no sensible definition of control-flow integrity for REINS to enforce.

As such, REINS and CFI have fundamentally different goals. CFI's goal is to micro-manage behavior within an untrusted binary to prevent attackers from corrupting its internal flows. In contrast, REINS' goal is to protect the environment outside the untrusted binary, not its internals. This includes external resources like the file system and network, and the trusted libraries that access them (e.g., OS/kernel libraries). The only flows that affect such resources are those that exit the untrusted code. For these, there are sensible, well-defined (but coarser-grained) control-flow policies apart from source code. For example, flows to the stack or data are disallowed (to block code-injection attacks), and flows to trusted libraries must obey the library's interface (e.g., its export address table).

REINS prevents these forms of malicious behavior based on the security policy. It is possible for an attacker to craft ROP [33] or Q [31] shell code to overwrite the stack pointer and break the internal control flows, but the attacker must ultimately manipulate the arguments of system calls to effect damage outside the confines of the untrusted module. These malicious system calls are detected and prevented by REINS.

6.2 Code Conventions

Our rewriting algorithm in §3 preserves the behavior of code that adheres to standard, compiler-agnostic x86 code generation conventions. Code that violates these conventions can yield rewritten code that fails verification or fails to execute properly, but never verified code that circumvents the monitor. Nevertheless, the practicality of the approach depends on its ability to preserve the behavior of a large class of non-malicious code. Compatibility limitations of this sort have been a major obstacle to widespread adoption of much past SFI research.

Code pointers. REINS expects each code pointer used as a jump target by untrusted code to originate from one of five sources:

- a low-memory address drawn from the program counter (e.g., a return address pushed by a `call`),
- data that points to a basic block boundary,
- a code address stored in the IAT,
- a return address pushed by a trusted caller during a callback, or
- a return value yielded by the system's dynamic linking API.

As demonstrated by our experiments, these cover a large spectrum of real-world binary code. Nevertheless, there are some unusual cases that REINS still rejects. For example, a program that computes external library entrypoints instead of requesting them from the system's linker is incompatible with REINS, and will typically crash when executed. Addressing such limitations is future work.

Reliable Disassembly. Binaries generated by most mainstream compilers mix code and static data within the `.text` section of the executable. REINS relies upon a classification algorithm that heuristically distinguishes code from data [38]. If code is misclassified as data, that code is incorrectly omitted from the rewritten binary's code section. If data is misclassified as code that looks like a possible computed jump target, the rewriter might overwrite some of the data with tagged pointers as it constructs the lookup table (see §3.1). This can result in corruption of the static data. However, data misclassified as code without such targets just contributes harmless, dead code to the rewritten binary's code section. Heuristics that conservatively classify most bytes as code with few computed jump targets therefore tend to work well for our system.

Function entrypoints are readily identifiable in most binaries by the characteristic function prologues and epilogues that begin and conclude most function bodies. The few remaining computed jump targets are gleaned through the disassembler's code reachability analysis and a few pattern-matching heuristics that identify instruction sequences compiled from common source language structures (e.g., switch-case statements) that often compile to computed jumps.

In practice we found that for most non-malicious programs, IDA Pro's automatic binary analysis works well, accurately identifying all code (with some data harmlessly misidentified as code) and identifying all computed jump targets (with some code harmlessly misidentified as a computed jump target). Any missed targets are easy to identify and correct manually, since their omission causes the rewritten binary to crash at precisely the site of the misclassified address in the (now non-executable) old code segment.

Dense Computed Jump Targets. A more subtle assumption of the algorithm is that all computed jump targets in the original binary are at least $w + 1$ bytes away from the next computed jump target or following data, where w is the system word size. This is necessary to ensure sufficient space for the rewriter to write a tagged pointer at that address without overwriting any adjacent pointers or data. Entrypoints packed closer than this are rare, since most computed jump targets are 16-byte aligned for performance reasons, and since all binaries compatible with *hotpatching* have at least $w + 1$ bytes of padding between consecutive function entrypoints [21].

In the rare case that two targets are within w bytes in the original code, the rewriter strategically chooses the address of the rewritten code section so that the encodings of tagged pointers into it can occasionally overlap. For example, with tag byte $t =$ `0xF4`, the sequence `F4 00 F4 00 04 00 04` encodes two overlapping, little-endian tagged pointers to addresses `0x0400F400` and `0x04000400`. By positioning the rewritten versions of these two functions at those addresses, the rewriter can encode overlapping pointers to them in the lookup table. With chunk size $c = 16$ and memory division $d = 2^{28}$, a rewritten code base address of $2^{24}(t \,\&\, 0\text{xF}) + 2^{16}t$ supports at least 15 two-pointer collisions and 1 three-pointer collision per rewritten code page—far more than we saw in any binary we studied.

6.3 Other Future Work

The experiments reported in §5 focus on testing the soundness, transparency, and feasibility of our static binary rewriting algorithm on a real-world OS, and on demonstrating the enforcement of some simple but useful security policies. Past work [17, 19, 28] has shown that IRM systems are capable of enforcing more sophisticated temporal properties when equipped with more powerful event languages and responses to impending policy violations that go beyond mere program termination. Developing policy-enforcement libraries that implement such policies is therefore a logical next step toward applying our framework to interesting, practical security problems for these real-world systems.

7. RELATED WORK

REINS is related to SFI, whose works can be divided into (1) source-level approaches, which instrument untrusted code with dynamic security guards at compile-time, (2) binary-level approaches,

which secure untrusted code at a purely binary level, and (3) system-level approaches, which secure the software at system call level. Table 4 summarizes and compares the major feature differences of the related works mentioned below.

Source-level Approaches. Most SFI implementations target source code and therefore insert security guard instructions at compile-time. Examples include StackGuard [8], DFI [5], WIT [2], BGI [6], G-Free [23], and CFL [4]. Source-level approaches differ significantly from the problem of securing COTS native code because a compiler typically has full control over the structure of the binary it generates, its pointer representations, and its implementation of computed jumps. In contrast, SFI systems for legacy native code cannot statically distinguish code pointers from data, recover control-flow or data-flow graphs reliably, or detect all instruction aliasing. Enforcing SFI without this information introduces many challenges.

The primary disadvantage of source-level approaches is their reliance on the support of a cooperating code-producer, who must (re)compile the untrusted or insecure code using a special compiler. Such cooperation is not a reasonable expectation for many classes of untrusted code, which are distributed as raw native code produced by arbitrary compilers, and that target mainstream system APIs such as Microsoft Windows or Linux.

Binary-level Approaches. In contrast, binary-level approaches require less compiler cooperation. They can be further divided into those that operate dynamically and those that operate statically:

Dynamic binary approaches use dynamic binary translation (e.g., Vx32 [13], Strata [32], Libdetox [25]), program shepherding [18, 3], or safe loading (e.g., TRuE [26]) to dynamically copy and instrument untrusted code into a sandbox at runtime. Any flows that attempt to escape the sandbox recursively re-trigger the copying process, keeping all untrusted, reachable code within the sandbox.

In contrast, static binary approaches in-line guard instructions into untrusted binary code statically before the code executes, and do not perform any code generation or translation at runtime. The only SFI systems other than REINS that target legacy, untyped, native code binaries to our knowledge are CFI [1]/XFI [10], PittSFIeld [20], NaCl [39], and SecondWrite [34]. CFI/XFI achieves reliable disassembly by consulting PDB files, which contain debugging information. The debugging information reveals important structural and typing information from the application source code without disclosing the source code text; however, PDB files are only produced by Microsoft compilers, and most code-producers do not disclose them to the public. This significantly limits the domain of binaries to which CFI/XFI is applicable. PittSFIeld and NaCl are similarly limited—PittSFIeld only supports `gcc`-produced assembly code and NaCl requires untrusted code to be (re)compiled by their tool chain.

SecondWrite tackles the problem of rewriting COTS binaries without debug or relocation metadata, but it does not support formal machine-verification, has not yet been applied to realize complete fault isolation, and is not yet mature enough to rewrite full-scale COTS applications [24].

Dynamic vs. static approaches have historically suffered a compatibility vs. performance trade-off. That is, the dynamic approaches can currently handle a much larger class of binaries than the static ones, including large-scale COTS applications, but at the expense of significant runtime overheads (e.g., 70% slowdown in Strata [32]). In addition, dynamic SFI systems are difficult to formally verify, and cannot be deployed on architectures that prohibit runtime code generation or that lack the hardware-level VM support that is often necessary to achieve reasonable performance.

In contrast, static approaches offer much lower overheads and formal, machine-checkable proofs of safety, but currently support only a very restricted set of binary programs that do not include most COTS applications. Therefore, REINS is the first purely static binary SFI system capable of supporting nearly arbitrary, large-scale, COTS Windows applications produced by mainstream compilers, including those that contain computed jumps, dynamic linking, and event-driven OS callbacks.

Table 4: Summary of related works. Symbol $*$ represents a limited feature for that work.

System	no source needed	no metadata needed	supports COTS binaries	handles computed jumps	handles callbacks	non-system approach	no kernel privileges	automated verification	stack exploit protection	system call interposition	enforces API policies	control-flow integrity	source available
StackGuard [8]						✓	✓		✓				✓
DFI [5]				✓	*	✓	✓		✓				
WIT [2]				✓		✓	✓		✓			✓	
BGI [6]				✓	✓				✓			✓	
G-Free [23]				✓		✓	✓		*				
CFL [4]				✓		✓	✓		*			*	
Vx32 [13]	✓	✓	✓	✓	✓	✓	✓			✓	✓		✓
Strata [32]	✓	✓	✓	✓	✓	✓	✓			✓	✓		
Libdetox [25]	✓	✓	✓	✓	✓	✓	✓		*	✓	✓		✓
Shepherd [18]	✓	✓	✓	✓	✓	✓	✓		*	✓	✓		
TRuE [26]	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓		✓
CFI [1]	✓		*	*	*	✓	✓		✓	✓	*	✓	
XFI [10]	✓		*	*	*	✓	✓	✓	✓	✓	✓	✓	
PittSFIeld [20]	*			*		✓	✓	✓	✓	✓			✓
NaCl [39]	*			*		✓	✓	✓	✓	✓	✓		✓
2ndWrite [24]	✓	✓	*	✓		✓	✓		✓				
REINS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		[29]
Janus [35]	✓	✓	✓				✓			✓	*		✓
SysTrace [27]	✓	✓	✓							✓	*		✓
Ostia [15]	✓	✓	✓							✓	*		

System-level Approaches. There are also many system-level approaches, such as Janus [35], SysTrace [27], and Ostia [15]. These use system call interposition to enforce policies that prevent abuse of the system API.

Unlike binary rewriting approaches, system-level approaches are transparent to the binary code. However, they cannot block attacks of one module upon another within the same address space, they cannot be deployed as a service (because the full implementation must reside on the client machine), and they can introduce compatibility problems, such as incorrect replication of OS semantics [14].

8. CONCLUSION

We have presented the design, implementation, and evaluation of a new SFI/IRM system, REINS, that monitors and restricts Windows API calls of untrusted native x86 binaries for which source code and debugging information are unavailable. The binary rewriting algorithm supports many difficult binary features, including computed jumps, dynamic linking, interleaved code and data, and OS callbacks, all without any explicit cooperation from code-producers, and it is behavior-preserving for a large class of COTS binaries. To the best of our knowledge, no past binary rewriting-based SFI work has achieved this. The enforcement mechanism requires no kernel extensions or privileges, making it applicable to shared and closed computing environments, and separate, light-weight machine-verification keeps the TCB small. Experiments on a number of COTS and malware programs show the effectiveness of REINS, and demonstrate that although rewriting doubles file sizes on average, runtimes increase by only about 2.4% and the median process size increases by only about 15%.

9. REFERENCES

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Information and System Security*, 13(1), 2009.

[2] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *Proc. IEEE Sym. Security and Privacy*, pages 263–277, 2008.

[3] P. Bania. Securing the kernel via static binary rewriting and program shepherding. http://piotrbania.com/all/articles/pbania-securing-the-kernel2011.pdf, 2011.

[4] T. Bletsch, X. Jiang, and V. Freeh. Mitigating code-reuse attacks with control-flow locking. In *Proc. Annual Computer Security Applications Conf.*, pages 353–362, 2011.

[5] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *Proc. USENIX Sym. Operating Systems Design and Implementation*, pages 147–160, 2006.

[6] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast byte-granularity software fault isolation. In *Proc. ACM Sym. Operating Systems Principles*, pages 45–58, 2009.

[7] P. M. Chen and B. D. Noble. When virtual is better than real. In *Proc. Workshop Hot Topics in Operating Systems*, pages 133–138, 2001.

[8] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. USENIX Security Sym.*, 1998.

[9] G. Erdélyi. IDAPython: User scripting for a complex application. Bachelor's thesis, EVTEK University of Applied Sciences, 2008.

[10] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *Proc. Sym. Operating Systems Design and Implementation*, pages 75–88, 2006.

[11] Ú. Erlingsson and F. B. Schneider. SASI enforcement of security policies: A retrospective. In *Proc. New Security Paradigms Workshop*, 1999.

[12] A. Fog. *Calling Conventions for different C++ compilers and operating systems*. Copenhagen University College of Engineering, 2009.

[13] B. Ford and R. Cox. Vx32: Lightweight user-level sandboxing on the x86. In *Proc. USENIX Annual Technical Conf.*, pages 293–306, 2008.

[14] T. Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In *Proc. Network and Distributed System Security Sym.*, 2003.

[15] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *Proc. Network and Distributed Systems Security Sym.*, 2004.

[16] K. W. Hamlen, V. Mohan, and R. Wartell. Reining in Windows API abuses with in-lined reference monitors. Technical Report UTDCS-18-10, U. Texas at Dallas, 2010.

[17] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *ACM Trans. Programming Languages and Systems*, 28(1):175–205, 2006.

[18] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure execution via program shepherding. In *Proc. USENIX Security Sym.*, pages 191–206, 2002.

[19] J. Ligatti, L. Bauer, and D. Walker. Run-time enforcement of nonsafety policies. *ACM Trans. Information and System Security*, 12(3), 2009.

[20] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *Proc. USENIX Security Sym.*, 2006.

[21] Microsoft Corporation. Using hotpatching technology to reduce servicing reboots. *TechNet Library*, 2005. http://technet.microsoft.com/en-us/library/cc787843.aspx.

[22] G. C. Necula. Proof-carrying code. In *Proc. ACM Principles of Programming Languages*, pages 106–119, 1997.

[23] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-Free: Defeating return-oriented programming through gadget-less binaries. In *Proc. Annual Computer Security Applications Conf.*, pages 49–58, 2010.

[24] P. O'Sullivan, K. Anand, A. Kotha, M. Smithson, R. Barua, and A. D. Keromytis. Retrofitting security in COTS software with binary rewriting. In *Proc. Int. Information Security Conf.*, pages 154–172, 2011.

[25] M. Payer and T. R. Gross. Fine-grained user-space security through virtualization. In *Proc. ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environments*, pages 157–168, 2011.

[26] M. Payer, T. Hartmann, and T. R. Gross. Safe loading – a foundation for secure execution of untrusted programs. In *Proc. IEEE Sym. Security and Privacy*, pages 18–32, 2012.

[27] N. Provos. Improving host security with system call policies. In *Proc. USENIX Security Sym.*, 2003.

[28] M. Rajagopalan, M. A. Hiltunen, T. Jim, and R. D. Schlichting. System call monitoring using authenticated system calls. *IEEE Trans. Dependable and Secure Computing*, 3(3):216–229, 2006.

[29] Reins source code. http://sourceforge.net/projects/x86reins.

[30] F. B. Schneider. Enforceable security policies. *ACM Trans. Information and Systems Security*, 3(1):30–50, 2000.

[31] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit hardening made easy. In *Proc. USENIX Security Sym.*, 2011.

[32] K. Scott and J. Davidson. Safe virtual execution using software dynamic translation. In *Proc. Annual Computer Security Applications Conf.*, pages 209–218, 2002.

[33] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proc. ACM Conf. Computer and Communications Security*, pages 552–561, 2007.

[34] M. Smithson, K. Anand, A. Kotha, K. Elwazeer, N. Giles, and R. Barua. Binary rewriting without relocation information. Technical report, U. Maryland, 2010.

[35] D. A. Wagner. Janus: An approach for confinement of untrusted applications. Master's thesis, U. California at Berkeley, 1999.

[36] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proc. ACM Sym. Operating Systems Principles*, pages 203–216, 1993.

[37] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proc. ACM Conf. Computer and Communications Security*, 2012. in press.

[38] R. Wartell, Y. Zhou, K. W. Hamlen, M. Kantarcioglu, and B. Thuraisingham. Differentiating code from data in x86 binaries. In *Proc. European Conf. Machine Learning and Principles and Practice of Knowledge Discovery in Databases*, volume 3, pages 522–536, 2011.

[39] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *Proc. IEEE Sym. Security and Privacy*, pages 79–93, 2009.

Code Shredding: Byte-Granular Randomization of Program Layout for Detecting Code-Reuse Attacks

Eitaro Shioji, Yuhei Kawakoya, Makoto Iwamura, Takeo Hariu
NTT Secure Platform Laboratories, NTT Corporation
9-11, Midori-Cho 3-Chome, Musashino-Shi, Tokyo 180-8585, Japan
{shioji.eitaro, yuhei.kawakoya, makoto.iwamura, takeo.hariu}@lab.ntt.co.jp

ABSTRACT

Code-reuse attacks by corrupting memory address pointers have been a major threat of software for many years. There have been numerous defenses proposed for countering this threat, but majority of them impose strict restrictions on software deployment such as requiring recompilation with a custom compiler, or causing integrity problems due to program modification. One notable exception is ASLR(address space layout randomization) which is a widespread defense free of such burdens, but is also known to be penetrated by a class of attacks that takes advantage of its coarse randomization granularity. Focusing on minimizing randomization granularity while also possessing these advantages of ASLR to the greatest extent, we propose a novel defensive approach called *code shredding*: a defensive scheme based on the idea of embedding the checksum value of a memory address as a part of itself. This simple yet effective approach hinders designation of specific address used in code-reuse attacks, by giving attackers an illusion of program code shredded into pieces at byte granularity and dispersed randomly over memory space. We show our design and implementation of a proof-of-concept prototype system for the Windows platform and the results from several experiments conducted to confirm its feasibility and performance overheads.

1. INTRODUCTION

Attacks that alter the control flow of a running program by corrupting pointers on memory have been a major threat of binary-layer software for many years and are still prevalent today. Such type of attacks is an attractive choice for attackers because they often give them chance to run arbitrary computation on the target host upon success. Typically, these attacks are conducted by somehow modifying the code-pointing address to point to the code the attacker wishes to execute. Traditionally, the code to be executed has often been prepared by injecting it into writable memory regions such as stack or heap using vulnerabilities such as buffer overflows. However, as code injection has become increasingly harder to perform with the spread of data execution prevention features, code-reuse attacks which jumps instead to code already existing on memory has become more popular. Examples of code-reuse attacks include the classic return-to-libc attacks as well as state-of-the-art variants of return-oriented programming attacks.

To counter this threat, numerous defenses have been proposed, and some of which are often categorized as control-flow integrity checking, sandboxing, software diversification, and so on. Apparently, however, most of them have not been very successful at being widely adopted for general use. This is most likely due to their strict restrictions which hinder their smooth integration into the development and deployment processes of target software. An example of such restriction is a requirement for recompilation with a customized compiler, which requires the proposal to be adopted by major compilers if it were become a standard. Another major problem is the integrity issues caused by changing the original instructions to be executed. This topic is discussed more, for example, in the literature[13] which describes the difficulties faced by software diversification approaches for massive-scale software distribution.

Nevertheless, there is one notable exception among these, namely ASLR(address space layout randomization). ASLR's main idea is to randomize the locations of various important memory objects, which include the load addresses of program images. As a result, it makes attackers much more difficult to locate the desired fragment of code to which they want to change the control flow, thereby securing against attacks that specify absolute addresses. ASLR was first introduced as a third-party Linux patch by the PaX project[28], and has recently been aggressively adopted by popular OSes such as Windows[1], Linux and Mac OS, and also by performance-sensitive mobile platforms such as iOS and Android[2].

One of the most contributing factors for the success of ASLR is most likely its simple nature, imposing surprisingly weak requirements on the target software; basically, it only shifts the location of a program image on memory thus its contents are kept unchanged besides some addresses requiring relocation. Also, it is notable that the sequence of instructions to be executed is not changed at all, which preserves program semantics perfectly. However, ASLR is known to be vulnerable against some type of attacks, such as attacks combined with memory address disclosures, or partial overwrite attacks[6]. This is due to the fact that ASLR has a coarse randomization granularity but this property is so closely related to its advantageous properties that it is intrinsically difficult to fix.

Based on this observation, by focusing on minimizing randomization granularity while also possessing these advantages of ASLR to the greatest extent, we propose a novel defensive approach called *code shredding*, a defensive scheme based on a simple idea of embedding the checksum value of a memory address as a part of itself. Code shredding provides randomization at byte-granularity, or in other words the finest randomization granularity possible, and is also expected to minimize recompilation requirements and incur no changes to the executed instruction sequence. For avoiding con-

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

fusion, it is worth emphasizing first here that code shredding does not *actually* shred code. This approach relies on combination of code replication and destination restriction of control transfer instructions, to achieve pseudo-dispersion of code. The fundamental idea of our approach is based on *self-validating addresses*, which encodes addresses by calculating the checksum of an address and embed the result *within* the address itself. With an appropriate design and implementation, a defense using this approach will give an attacker an illusion that the code to which they wish to transfer the control flow is shredded into pieces at byte granularity and are randomly dispersed over memory space. Our proposal is not without any drawbacks of course, and we admit that it may incur considerable amount of runtime overhead on time and space. Though we are still in a status of seeking for ways to alleviate these overheads, we make a proposal of an additional sub-technique called *code mirroring*.

We have implemented a prototype of this proposal for the Windows platform using the Pin dynamic binary instrumentation framework. Experiments are conducted to evaluate its security effectiveness and performance. The contributions presented in this paper are summarized as follows:

- We present *code shredding*, a novel scheme for detecting invalid control-flow transfers, based on the following two ideas:
  - *self-validating address*: allows verification of memory addresses by encoding addresses in such a way that its checksum value is embedded as a part of itself.
  - *code mirroring*: allows an encoded address to be treated as a real addresses by actually mapping multiple copies of the image on memory.
- We present our design and implementation of a proof-of-concept prototype of code shredding system for the Windows platform. The results of our brief experimental evaluation suggests its feasibility concerning security and compatibility, and also clarifies some problems which need to be addressed in the future.

The rest of this paper is organized as follows: Section 2 describes the problem, Section 3 and 4 explains in detail the design and implementation of our proposal, Section 5 describes the evaluation procedure and the result, Section 6 discusses our limitations and related researches, and finally Section 7 concludes the paper.

2. PROBLEM DESCRIPTION

In this section we describe the type of attack we focus on, investigate the problems confronted by existing defense approaches, and state the problems which we aim to solve in this paper.

2.1 Control-Flow Hijacking Attacks

The type of attack which we focus on is what changes the control flow of a running program by modifying code pointers. We define a code pointer as any data on memory that is meant to contain an address which points to, or will eventually point to, a location on memory where executable instruction exists or will exist. Examples of code pointers include return addresses of function calls or function pointers. The goal of attackers is to execute their desired code which is a malicious sequence of instructions that triggers an attack. This can be achieved by either code injection or code reuse. Code injection writes to memory the code prepared by the attacker. However, as defensive measures such as NX-bit or DEP are becoming more common, code injection is becoming harder to perform. Code reuse, on the other hand, is not affected by such defenses because it uses already-existing codes on memory with execute permission. A famous example of such attack is return-to-libc attack[4], but recently a more generalized form of such attack called ROP (Return-Oriented Programming)[5] has become a large threat and is evolving continuously, with its variants or automation techniques being constantly presented.

2.2 Existing Defenses: ASLR

Although ASLR makes it much more difficult to write stable exploits, it is not a perfect measure even when combined with DEP or other defenses. It is in fact possible to evade ASLR when certain conditions are met. In this paper we focus on the two types of attacks that defeats ASLR, namely, attacks using address disclosure and partial overwrite attacks. ASLR-bypass using address disclosure is becoming very common, while partial overwrite attacks are relatively not so common due to its difficulty of application. However, because it shares the same intrinsic problem with address disclosure, and it in fact has been used for exploiting the famous ANI vulnerability[25], we refer to it as our motivating example. Also, as the number of systems hardened with ASLR increases, advanced exploitation techniques like partial overwrite may start to become more common in the future. Next, we describe these two attacks in more detail.

2.2.1 Attacks with Address Disclosure

Memory disclosure is a software flaw such that the content of memory is leaked to the attacker. It may occur, for example, when a format string vulnerability is exploited. Address disclosure is its special case, where a pointer is disclosed to the attacker which becomes a problem when combined with other vulnerability such as buffer overflow that can write to code pointers. Let a represent the address where instruction A is located on memory, and let x represent the address of an arbitrary instruction X in the same image. Under the assumption that the attacker has the same program image as the target program, upon knowing the address a, the attacker can calculate $x = a + d$ using the relative distance d between A and X derived from the attacker's own copy of the program image. Thereby the attacker is able to determine the address of an arbitrary instruction in this program image.

2.2.2 Partial Overwrite Attacks

On the other hand, a partial overwrite attack does not even require disclosure of concrete address values. All it requires is the knowledge that the overwriting target is an address pointing to a code in the program image. The idea behind this attack is that a typical ASLR implementation only randomizes the upper bits of address (e.g., Windows 7 32-bit randomizes bits 17 - 24) likely due to memory allocation restriction or performance reasons. Therefore, even if the upper bits are randomized, by overwriting the lower bits of an existing address, the attacker is able to specify an address within the possible range. Figure 1 describes an example of such attack. Here the stack holds a return address pointing to 0xKKR12340 where R denotes the byte randomized by ASLR. Traditionally, ASLR has been effective in such a way that, if the attacker does not know the value of R, this byte cannot be overwritten, and therefore the whole address cannot be overwritten. However, the attacker can still overwrite the lower bytes to specify addresses of program image in the shown range without modifying R, and make the return address point to an address in range 0xKKR10000 - 0xKKR1FFFF without any knowledge of R. Note that this is feasible because the byte order of IA-32 is little endian, i.e., an address is stored on memory lower bytes first.

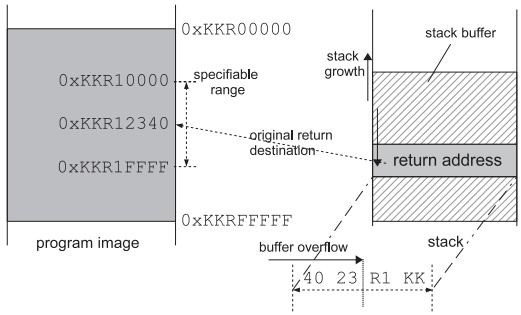


Figure 1: partial overwrite attack

2.2.3 ASLR Problems

One of the root causes the attacks described above are valid against ASLR is that the attackers are able to relate an address to another by referring to the same program image of their own. One trivial way to prevent them is by making the software publicly unavailable, since these attacks must assume that the attacker possesses the program image, but such measure is unrealistic in most cases. Here we define the *granularity* of randomization as the size of program content whose location on memory can be uniquely determined from an address disclosed to the attacker. ASLR provides randomization granularity at the size of program image, because any leaked address within the code immediately reveals the address of an arbitrary code on the image. In this sense, having finer granularity is more preferable because it will limit the attacker's ability relate addresses together, and ideally, having *byte granularity* will make such attacks impossible.

ASLR also carries several other problems. One of such is not having enough randomization entropy on systems with 32-bit memory space, making it vulnerable against brute force attacks[14] in some cases. Also, many of the ASLR-bypass exploits in the wild to date leverage the fact that ASLR is often an opt-in defence, and reuses code on ASLR-disabled modules. Brute-forcing is likely to be mitigated by the future transition to 64-bit systems, and regarding the opt-in problem, non-ASLR enabled binaries are on the decrease as observed in the case where Firefox is no longer accepting plug-ins not supporting ASLR[26]. It is also worth noting that Microsoft's EMET, which is a security add-on package for Windows including ASLR-forcing feature, has recently been officially supported[27], which implies that ASLR is now applicable to certain software not previously assured to be compatible with it. However, the problem with address-relation is an intrinsic property of ASLR, and is unlikely to be mitigated by such improvements in the future, suggesting the need of exploration for post-ASLR measures.

2.3 Existing Defenses: Other Defenses

Since the problem of control-flow hijacking has been around for a long time, there are numerous other approaches for defending against it. For example, there are approaches for randomizing at finer granularity such as at function granularity[8], but it still suffers from address disclosure attacks. There are also other approaches that offer randomization at finer level[9, 10, 11, 12] such as at instruction level, but they generally require recompilation from the source code, or changes executed instructions which may cause integrity problems. Some of these are discussed and compared to our proposal in Section 6.

2.4 Problem Summary

We conclude this section by describing the threat model and the properties which we aim to incorporate into our proposal.

The Threat Model:

- The execution environment is assumed to have data execution prevention system or equivalent enabled, or in other words all writable regions on memory are non-executable.
- During execution, the attacker somehow successfully overwrites a part or whole of the content of a code pointer to an arbitrary value.
- Prior to the rewriting, the attacker is able to obtain a leaked address within the program image. It can be any address except for the address overwritten to, mentioned above.

Desired Properties:

- Security: Under the threat model presented above the ability to infer the location of instruction to execute from the leaked address should be limited to the greatest extent, i.e., having the finest randomization granularity possible.
- Compatibility: The requirement for recompilation should be minimized, while preserving the integrity of the original program as much as possible.

Note that other aspects not listed above such as those concerning the effects on execution environment or performance will not be our main focus in this paper although they will be mentioned briefly.

3. DESIGN

To solve the problems stated above, we propose a novel defensive approach, *code shredding*, which detects invalid control-flow transfers caused by code pointer modification. In this section we describe its overview and design. Briefly described, this scheme consists of (1) conversion of addresses into specially-encoded *self-validating addresses*, and (2) validation of target addresses used in control flow transfer instructions at their execution. Below we describe these procedures in more detail.

3.1 Self-Validating Address

A self-validating address is simply generated in the following way:

- perform a checksum calculation on the bits at the fixed position of the address (we refer to these bits as *input bits*), and
- place the checksum value as the bits at another fixed position of the same address (similarly, *checksum bits*).

An address is considered valid if the calculated checksum from the input bits matches the checksum bits. It is called self-validating because the information needed to validate an address is self-contained, as opposed to other methods that must compare the address with other(e.g., saved) values. Unless the attacker knows how to perform the computation, it is impossible for them to generate, with enough success probability, a valid address that can be used for code-reuse attacks. In addition, if it is impossible to infer the checksum value of any other input value from a given valid converted address, it provides byte-granularity randomization. This can be realized, for example by using a secure hash function which satisfies such property and feeding the input value combined with a random value which is hidden from the attacker.

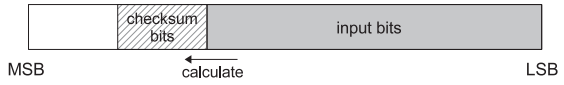


Figure 2: a self-validating address

Some design choices for self-validating addresses include the position and length of the input and checksum bits. For this proposal we place the checksum bits at the position higher(more significant) than input bits, and make checksum bits shorter than input bits, as illustrated in Figure 2. The actual length and position are configured considering the size of available memory space, the size of program image, and the level of security to assure.

3.2 Conversion

To be able to distinguish invalid addresses from valid addresses, the valid addresses must be converted in advance. There is no point of converting an address after it has been modified by an attacker, so it is desirable to convert it as soon as the address is generated. There are basically two types of address generation to consider: addresses generated before program execution starts, and those dynamically generated after.

For those generated before, the loading of program image must be hooked, and absolute addresses that exist in the image must be located and converted. The locating of such addresses is possible using the relocation information on the executable. This implies that our proposal assumes the existence of such side information, but this is the same condition assumed by ASLR, and is relatively easy to satisfy thanks to the widespread of ASLR.

Addresses dynamically generated during program execution are mainly return addresses from function calls, so call instructions must be hooked and the return address written to the stack must be converted. There are some other occasions, mostly application or loader specific, where absolute addresses are dynamically generated which will be discussed later.

3.3 Validation

Validation is done just before the execution of a control flow transfer instruction. It calculates the checksum value in the same manner as it is done in conversion and compares the calculated value with the checksum value embedded in the address. If they match, the address is considered valid, and otherwise it is considered invalid. Upon detection of an invalid transfer, the execution can be halted or be continued depending on the policy. To continue execution after validation, the address must be converted back to the original address where the actual code exists, but this procedure can be omitted as described later.

3.4 Segments

For the following discussions, and also to make the above description more comprehensible, we introduce the notion of *segments*. As the result of applying this design decision, letting N_{inp} be the length of input bits and N_{chk} be the length of checksum bits, we observe $2^{N_{chk}}$ consecutive blocks of memory regions, each block having the size of $2^{N_{inp}}$ bytes. We will refer to these blocks as *segments*. Each segment must be at least the size of the original program image, for each byte in segment corresponds to each byte of original program image. Depending on the value of checksum, any valid address will belong to one of these segments.

Increasing the length of checksum bits increases the security strength because addresses are distributed over larger space, lowering the success probability of guessing attacks. The pattern of dispersion which designates which address belongs to which segment, can be randomized by feeding a random value as a part of its input. By choosing a new random value, say, for every process, the valid code will appear to be located in random segments from the perspective of attackers, making them difficult to specify them in their exploit codes, as shown in Figure 3.

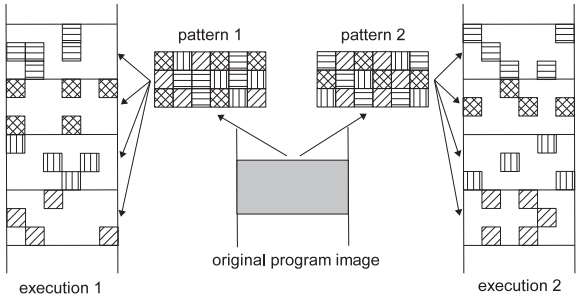


Figure 3: two patterns of specifiable destination addresses dispersed across multiple segments, randomized differently in each execution (each square represents a byte).

Example.

Here we show an example, with its layout illustrated in Figure 4. Assuming 32-bit memory address space, let an address be represented by 0xKKAXXXXX where the input bits are bits 1 – 20 denoted by X'=XXXXX, checksum bits are bits 21 – 24 denoted by A, and remainder bits are bits 25 – 32 represented by K which are don't-care bits in this example. Let Hash() be a hash function from 20 bits space to 4 bits space, and R be a random constant chosen uniformly random from 20 bits space. We define an address valid if it satisfies A = Hash(R + X'). Then any address originally existed in the program will, after conversion, belong to one of the 16 segments 0xKK000000 – 0xKK0FFFFF, 0xKK100000 – 0xKK1FFFFF, ⋯, 0xKKF00000 – 0xKKFFFFFF depending on the calculated value A. For example, any address satisfying A = Hash(R + X') = 1 would belong to 0xKK100000 – 0xKK1FFFFF.

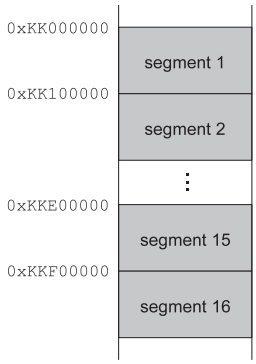


Figure 4: memory layout of the given example

3.5 Code Mirroring

With the approach above, a converted address does not actually point to an executable code and thus requires that, after validation, the address be converted back to point to the segment where code

actually exists. Here we propose an extensive idea, *code mirroring* for eliminating the need for this by actually placing executable code in each segment, letting the execution continue at the destination pointed by the converted address. Figure 5 illustrates the effect on execution flow when code mirroring is introduced, and diagram (c) shows dotted arrows and solid arrows overlapping with each other because now converted addresses can be treated as a real address *unmodified*. Consequently, the control flow will proceed in an interesting manner; a switch to a random segment occurs upon every execution of monitored control transfers instructions.

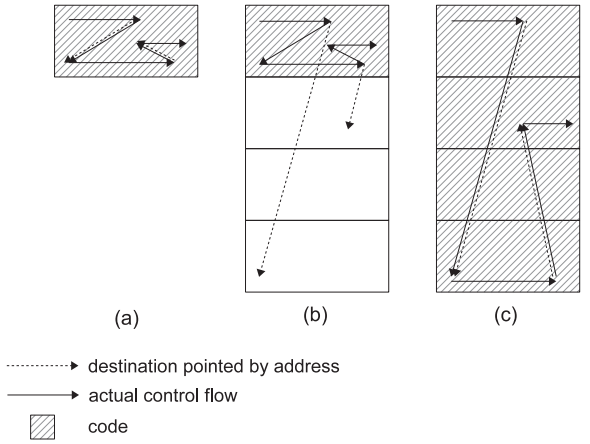


Figure 5: (a) normal execution, (b) code shredding without code mirroring, (c) code shredding with code mirroring

An obvious drawback of this scheme is the additional use of memory space, which is non-negligible because it increases exponentially with the number of checksum bits, or linearly with the number of segments created for each image. To alleviate this overhead, we believe that intra-process memory mapping which allows mapping of identical objects on virtual address space to the same physical memory space (Figure 6). This may be implemented with system APIs, such as mmap on Linux or MapViewOfFile on Windows. While such APIs are usually used for file mapping or sharing memory contents among different processes, here we use it within the same virtual address space of a single process. Note that an idea similar to intra-process memory mapping has been proposed as VMA Mirroring[30], as discussed later.

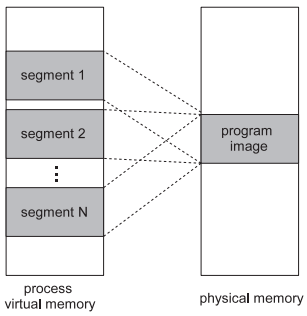


Figure 6: intra-process memory mapping

3.6 Overall View

Lastly, we show the overall flow of our system in Figure 7, for the case with single loaded image. Note that the procedures p1 and p2 enclosed by dotted lines are in trade-off relations, i.e., when code mirroring is applied p1 is required but p2 is not, and vice versa.

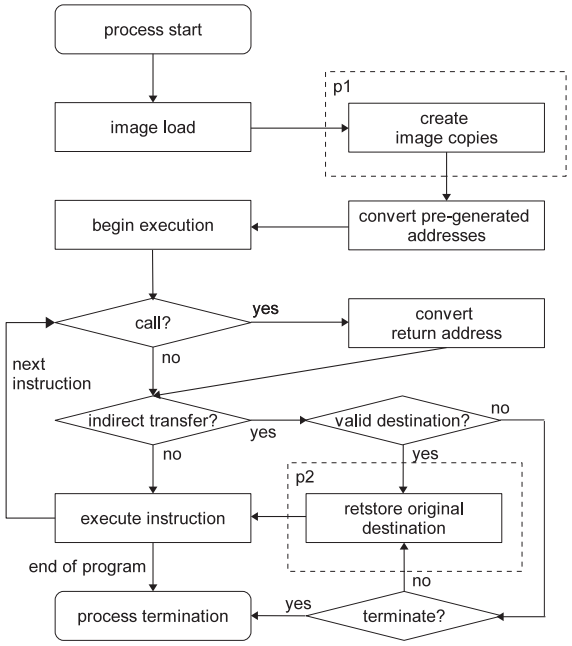


Figure 7: flowchart of procedures of our scheme.

4. IMPLEMENTATION

In this section we describe the prototype implementation of our proposal. It is implemented for the Windows/IA-32 platform using the Intel Pin dynamic binary instrumentation framework[7]. Pin is a process-level VMM that JIT compiles code as the targeted program executes, and allows instrumentation of additional code. Note that our implementation is used for a brief evaluation in this paper and is still an ongoing work. Some functions such as intra-process memory mapping has not been implemented yet, but it should not greatly affect basic feasibility nor performance.

4.1 Conversion

Pre-Generated Addresses

We hook the end of program image loading, and copy the loaded program image to each segments which are allocated. This is when we convert the pre-generated addresses. Generally, PE files have metadata called relocation table for finding absolute addresses hard-coded in the program image for relocation purposes, so we find these addresses by parsing the entries of the associated table which can be located from header information. Our conversion targets are code pointers, so in this implementation we make all data pointers in all segments point to the data section of the first segment. The decision to determine whether the address points to data or code is done by investigating the flags of the section of the destination. If it has a write flag, it is considered data and is not considered subject for conversion. This is shown to be not sufficient in some cases and will crash some programs, and this problem is discussed later.

Note that with this setup, we have a great property such that, although each program image is loaded at different memory locations

their contents will be exactly the same, which implies that it can be effectively shared by intra-process memory mapping.

Dynamically Generated Addresses

All CALL instructions generate a return address which is pushed onto the stack when they are executed, so these instructions must be hooked for address conversion. One straightforward way for doing this is by hooking the instruction followed by the CALL instruction, and directly modify the return address stored on the stack. For performance reasons, we instrumented code that emulates a call instruction that does the conversion at the same time, avoiding the need for the second memory access.

4.2 Validation

The destinations of all control flow transfer instructions that refer to absolute addresses are subject for validation. On IA-32, they are absolute JMP, absolute CALL, and RET instructions, so we set hooks on all of these instructions. Note that we do not need to consider relative control flow transfer instructions because such instructions all take immediate operand on this architecture which cannot be targeted by our attack model because they cannot be overwritten. After hooking, the destination address is calculated and then validated.

4.3 Loader-Specific Considerations

Our prototype implementation is for the Windows system, and because we made no modifications to the loader, there are some special considerations required where interactions with the loader occur. Some of the information about internals of Windows or its API specifications mentioned below can be referred to at the MSDN website[20].

- *Handles*: Windows uses data objects called handles to refer to system objects such as processes, files, and threads. For images (or modules) which are our main concern, their handle is essentially a pointer to the base address of the image, which implies that the value of handles are relocated depending on where the image is loaded. The system often uses a handle as an ID of an object, not as an address, so it would be problematic if handles are regarded as an address for our conversion, because then the system would not be able to recognize them. Therefore, when parsing through the relocation table to look for pre-generated addresses, if an address points to the base address, we leave it unconverted as it is most likely used as a handle. For the same reason, when some Windows APIs such as LoadLibrary or GetModuleHandle return handles, we leave them unconverted. Note that, on the other hand, it may have a side-effect causing a problem if handles are used as addresses.

- *DLL Entrypoints*: DLLs often (though are not required to) have an entrypoint function defined, which are called on creation/termination of threads and processes, and also on calls of LoadLibrary/FreeLibrary APIs to perform initializations and cleanups. The absolute addresses of these entrypoint functions are stored in the internal database of the loader, and they point to the image loaded at the original base address, causing false-positives when they are dispatched. Therefore, when image loading is hooked, we parse through the loader database which is accessible from PEB(Process Environment Block), and convert these entrypoint addresses as soon as they are registered.

- *GetProcAddress*: This API takes as its argument a module handle and a procedure name, returning the address of the specified procedure. However, the returned address turns out to be an unconverted address, and this is most likely because of how functions are exported in windows. The address is probably calculated by adding the base address of the image and the RVA(Relative Virtual Address), the offset from the base address, obtained from the export table. To avoid this problem we hook the returning from of this API and convert the return value.

- *GetModuleHandleEx*: This API is used to obtain a module handle from the argument pointing to the name of image name to look for. However, when it is called with the dwFlags argument with an appropriate bit set, it accepts an arbitrary address instead of a string pointer, and returns a handle of the module that contains this address. If the address passed to this API is a converted address, the loader would return a value indicating that the lookup has failed, causing a semantic change to the program, if not a crash. Therefore, we hook the call to this API, check the dwFlags, and if its 0x4 bit is set, we convert back the address specified as the argument lpModuleName.

- *Restriction of Pin*: Although this is not an issue directly related to the loader, we mention here that Pin has a limitation where it becomes active only after kernel32.dll, kernelbase.dll, and ntdll.dll have been loaded. Thus, it misses some of the initial instructions executed by these DLLs, as mentioned in the paper published by its developers[22]. This is most likely the cause for false negative and crashes when trying to apply our system to these DLLs. Therefore, in our prototype we will always be excluding these DLLs from our target.

4.4 Hash Calculation

For this prototype we used a simple hashing algorithm which just adds up each byte of the input value, adds a randomly generated byte value, and takes the modulo $2^{N_{chk}}$ of the result, where N_{chk} is the number of checksum bits. Note that such hash algorithm is not considered secure in practice, as two disclosed addresses would instantly reveal the secret random key, but for our threat model it should be enough. Also, for performance reasons we cached the pre-calculated hash values on a cache table which consumes memory size approximately equal to the memory size consumed by a set of segments created for one program image. Still, note that only one copy of this table is required, because it can be shared among all different images in the same process. Such table is feasible for 32-bit implementation, but considering that they fully consume the equal size of physical pages unlike images, and that the table size grows with 64-bit implementation, calculating hashes on-the-fly may be a better idea.

5. EVALUATION

In this section, we briefly evaluate our prototype from the two aspects we decided to focus on, which are security and compatibility. Other issues are discussed in the next section.

5.1 Set-up for Experimental Evaluation

We have conducted simple experiments with our prototype. The environment is Windows 7 Ultimate SP1 32bit, 2-core Intel Xeon X5670 at 2.93GHz with 4.00GB RAM on VMWare Workstation 7. It should be noted that our current implementation only partially supports application to DLL images, due to some unanalyzed crashes and restrictions of Pin. The number of input bits is 24, and

that of checksum bits is 2, implying 4 copied segments are created for each image loaded for the process.

5.2 Security Evaluation

5.2.1 Address Disclosures and Partial Overwrite

Even if the attacker has gained an address pointing to a code in the program image, without knowing the secret randomization key, attacker cannot identify which segment the address of target code exists. Guessing would only allow success at the rate of once every $2^{N_{chk}}$ attempts on average, where N_{chk} is the length of checksum bits. Partial overwrite would also not work for the same reason.

One may argue that, this scheme is not really byte-granular because a leaked address will reveal the checksum for all other addresses with the same input bits. In other words, using a leaked address 0xKKYAAAAA, where Y and AAAAA are checksum bits and input bits respectively, the attacker can designate all addreses 0xXXYAAAAA for arbitrary XX. However, in such a case, the related addresses are inter-modular, so that the leaked address and the addresses targeted for code reuse can safely be isolated by ASLR. The only concern is when the disclosed address turns out to match the destination address of invalid transfer. However, as ROP usually requires multiples addresses, one of each gadget, so such case should not be significant because the effect is confined to at most one.

5.2.2 Experimental Evaluation

We have also run an experiment to see if it detects partial overwrite attacks, although it is somewhat already obvious that it does. Assuming the case where there is an exploitable vulnerability in the function LoadLibraryEx of kernel32.dll which is called by Firefox 5.0.1 executable, we simulated a partial overwrite attack that modifies the lower 16 bits of the return address from this function, returning into code on the Firefox main executable. We confirmed that our prototype successfully detected this attack as expected.

5.3 Compatibility Evaluation

Here we present and analyze two examples of problematic cases we encountered during experiments, generalize and clarify the problems our proposal has, and suggest workarounds.

5.3.1 Experimental Evaluation

To test compatibility, we ran various programs on our prototype. Programs such as ftp.exe, calc.exe, mspaint.exe, and notepad.exe from the folder C:\Windows\System32, and also Adobe Reader 10.1.1(Non-protected mode), Firefox 5.0.1, Internet Explorer 8.0, Hover 6.1, etc. were confirmed to run without problems, except for some that needed a workaround mentioned in case analysis 2. Again, it should be noted that DLLs are not supported so perfect compatibility is not yet assured. Some of the problems confronted are analyzed below:

5.3.2 Case Analysis 1: Direct computation on addresses

The first example we have is a false positive case which was caused by the following instruction sequence.

```
mov ecx, [ptr]
add ecx, 5
call ecx
```

It loads a value from the memory to the ECX register, adds an immediate value to it, and then calls a function it points to. This will likely cause a false positive because the input bits of the address are modified, but the checksum value no longer corresponds to it. Our proposal compares the checksum calculated from lower bits with that of higher bits, so when an already-generated address is directly modified, it would cause a false positive. A workaround for this problem would be to detect such occurrences by static/dynamic analysis and take it into consideration when calculating a checksum. It still cannot cover all cases, so for a perfect solution, it is needed to restrict the compiler from generating such code.

5.3.3 Case Analysis 2: Mixture of data and code

Next, we present a case which caused a crash. It occurred in a function that takes two data pointers, one holds the start address of a function pointer table, and the other one holds the end address of the same table. In the function, each function on the table is executed by incrementing the starting pointer until the it reaches the end address. The problem arises because both of these addresses are subject to conversion because the table is located in the code section despite that they are actually data. Therefore, with high probability the two addresses would point to different segments, and the function will keep reading the table beyond its end as illustrated in Figure 8.

This is another example that illustrates a problem being caused by the difficulty of separating code pointers from data pointers. To overcome this problem, the compiler must add additional information on the relocation information indicating if the address points to data or code. For this particular case, the first and the last entries of the table turned to be always zero, so we evaded this problem by skipping the conversion of addresses which points to value zero, which is unlikely to be a valid code.

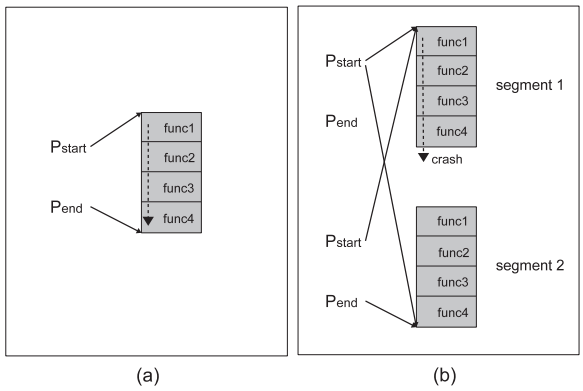


Figure 8: P_{start}, P_{end} are pointers to the head and tail of the function table respectively. (a) unmodified execution where each function pointer on the table is executed consecutively, and (b) execution with code shredding where a reference to function pointer occurs outside of the table.

5.3.4 Other Crashes

Although most programs we tested ran properly when code shredding is applied only to the main executable, there are still some DLLs that causes crashes when code shredding is applied. These cases are currently under analysis for their cause.

5.4 Runtime Performance

As our benchmarking application, we used Bzip2 1.0.5 for Windows[19], which is part of GnuWin toolsets, compiled from source

code with Visual Studio 2010. This application loads 5 images, bzip2.exe, msvcr100.dll, ntdll.dll, kernelbase.dll, kernel32.dll, and the last three DLLs are not targeted due to the restriction of Pin mentioned earlier in this paper. The executed command is: bzip2 -zkf rand_X.dat, where X represents the size of used data file in bytes and is one of 100K, 1M, 10M, 100M, and 1000M. The content of data file is randomly generated by the command: head -c X /dev/urandom > rand_X.dat.

For images not being targeted with code shredding, we hooked all call instructions to perform dummy conversion procedure, and also hooked all indirect control-flow transfer instructions jumping to non-targeted images to perform dummy detection procedure to artificially introduce the overhead.

Table 1: Execution time and overheads

	nat(s)	pin(s)	cs(s)	cs/nat	pin/nat	cs/pin
100K	0.077	1.159	1.998	26.10	15.14	1.72
1M	0.311	1.782	2.882	9.27	5.73	1.62
10M	2.590	7.763	11.46	4.43	3.00	1.48
100M	25.17	66.92	96.18	3.82	2.66	1.44
1000M	254.9	668.0	931.8	3.65	2.62	1.39

Table 2: Number of executed instructions and hook ratio

	num. of inst.	convert(%)	verify(%)
100K	1.25×10^8	0.165	0.168
1M	1.27×10^9	0.221	0.221
10M	1.27×10^{10}	0.227	0.228
100M	1.28×10^{11}	0.228	0.228
1000M	1.27×10^{12}	0.228	0.229

Table 1 shows the execution time and overhead of Bzip2 for compressing each data file. Columns 2-4 show the execution time in seconds, where nat: native execution, pin: pin without any instrumentation, and cs: code shredding. Columns 5-7 show their ratio, so for example, cs/nat indicates the overhead of code shredding over native execution, expressed as ratio. The results show that as the execution time grows, the cs/nat overhead converges to around 3.6, and cs/pin overhead converges to around 1.4. The large overhead with short execution is mainly because of how Pin works, including Pin's setup time and the initial execution JIT overhead, as shown by the pin/native Pin without instrumentation.

We also profiled the execution frequency of hooked instructions that cause overhead, i.e., the call conversion hooks (call instructions) and verification hooks(indirect control-flow transfers). Table 2 shows the percentage of such instructions executed out of all executed instructions. It shows that they both converge to around 0.228 percent for this application.

We have also ran some GUI applications mentioned in the compatibility evaluation part, and although booting took longer than usual, we were able to use them, (e.g. browse the web, edit texts) without much stress once booted. There are some Pin-dependent as well as Pin-independent optimizations which we have not yet applied, so these results are only tentative.

6. DISCUSSIONS

In this section we discuss the limitations of our work as well as related work.

6.1 Known Limitations

6.1.1 Non-Code Pointers

To change the control flow, the attacker may instead choose to modify data pointers such as flag data or user information which are referred to at conditional branching statements, but such attacks are out of scope of this paper. However, there is a case where data pointers can be indirect code pointers, which hold addresses of code pointers. This will become a problem because since data is all in the same segment, if we have an address disclosure of any of these data, all such indirect code pointers will be targeted for code-reuse attacks.

6.1.2 Memory Usage

Although consumption of physical memory space may be alleviated to a certain extent, consumption of *virtual* memory space may start to become an issue when larger number of copies is required to satisfy strict security demands. We expect that this problem will eventually be solved by the recent spread of 64-bit systems.

6.1.3 Dynamic Code Generation

Another type of dynamically generated addresses pointing to code, is seen in applications that generate and execute code on-the-fly as in script language interpreters. Since it is not possible to detect such generation of address without understanding the program semantics, our current approach is not applicable to such programs. This implies JIT spray attacks[21], which is a code reuse attack on dynamically generated code, cannot be prevented by our defense, although it can be claimed that JIT spray attacks are limited to special applications and such flaws should be fixed independently. In any case, for applying our proposal to programs with dynamically generated codes in general, it needs to be implemented at the application layer.

6.1.4 Dynamically Loaded Libraries

As stated in the previous section, our current implementation can target only one program image, which is the executable file image. Because most programs load tens of additional DLLs after the loading of main executable file, and because in many cases majority of executed instruction exist in DLLs, the effect of this may not be negligible. Also, the loading of DLLs is sometimes delayed until it is actually used, and because our proposal incurs considerable initial overhead, this may become a problem if it occurs too frequently during middle of execution.

6.1.5 Execution Environment

Because the Pin module itself cannot be the target of code shredding and it will be loaded to the same address space as the targeted software, the attacker may perform an attack by returning into Pin code. Therefore, it is more preferable to implement it in a more isolated layer such as the OS layer, the hypervisor layer, or even the hardware layer.

6.2 Related Works

Here we describe and compare some other research works, which share common ideas with our proposal.

Software Diversification

Software diversification changes the software and/or the execution environment in such a way that the attacker cannot succeed the attack unless he knows how it is changed. A well-known approach is instruction set randomization[3] which encodes instructions with a secret randomized key and decodes them right before execution

making code injected by the attacker not decodable. There are other approaches such as stack reversing[32], but while such approaches are valid for preventing code injection they do not prevent control flow hijacking. Our approach is similar to software diversification in the sense that similar idea is applied the addresses themselves, making overwriting, or injection of addresses impossible.

ROP prevention

The recent increase in the number and complexity of ROP(return-oriented programming) attacks have lead to many proposals specialized for defending against such attacks, for example by monitoring the frequency of return instructions[31]. Recently, a method to eliminate ROP gadgets by static conversion of binaries has been proposed[23], which seems to be a quite effective and practical measure against ROP. Our method is not limited to ROP attacks and can prevent a more broad range of attacks which includes return-to-libc, ROP without returns[29], and other type of control-flow hijacking attacks that may be invented in the future.

Pointer Encryption

The most similar, and perhaps the most competing approach to ours would be the defensive measures based on encryption of addresses[18]. While their security depends on a solid cryptographic base instead of probability, encryption tend to cause high overhead. Also, while our approach encodes an address in a form still directly usable by instructions, such encrypted addresses require decryption before usage.

Shadow Stack

Another well-known approach is to keep track of the return address on a shadow stack which emulates the real stack, and upon function returns compare the return address to the saved values as in, for example ROPDefender[24]. An obvious limitation of such approach, aside from requiring extra memory for shadow stack, is that it can only defend against attacks that modify return addresses. One might argue that other types of addresses can also be checked by extending shadow stack to keep the copies of them; however, such addresses do not always stay stay at the same location like return addresses, and they are moved and copied to other places such as heap, stack, and of course, registers. It is hard to track these without applying a technique such as taint analysis which is extremely performance-intensive. In contrast, in our scheme the information needed to validate addresses are *self-contained*.

Control Flow Integrity

Control flow integrity(CFI) is an approach that generates a model of valid control flow graph through static analysis of the source code or binary at run-time, and enforces it during execution. While it allows a more strict and deterministic defense, it incurs heavy runtime overhead. Also, they tend to suffer from false positives when analysis is not sufficient due to lack of debug information which is common on COTS software.

Other Defensive Approaches

Code Pointer Masking[15] is an approach which is similar to ours in the sense that it is a probabilistic approach that verifies destination addresses at runtime. Just before its use, a code pointer is applied a special mask generated beforehand by OR-ing valid destinations. Applying this mask scrambles the pointer only if it is not pointing to a valid destination. The disadvantage of this approach, besides requiring source code analysis, is that as the number of valid destinations increases, the mask becomes more tolerant thus allowing more possible false negatives. On the other hand, in our approach the security strength is independent of the program content, and is controllable to a certain extent, i.e., the security strength can be increased to a desired level by increasing the number of segments as long as memory allows.

In-OS address space layout randomization is studied in the recent work[33] and it is claimed that information leakage is a more serious issue in the kernel context which motivates our work. Also, their idea of live re-randomization may be applied to complement our system for further strengthening. Also, binary stirring[34] is another recent proposal of static binary randomization, where challenges of transformation is studied throughly, which may be inspected to further identify and fix our compatibility issues. While both of these works have advantageous features, they have randomization granularity of basic block.

Memory Sharing

The idea of mapping a physical page to multiple pages in virtual memory had also been proposed a while ago by the PaX project, as VMA mirroring[30] for Linux systems. The implementation details of VMA mirroring itself, however may be applicable when extending our work to Linux. Also, a similar idea in a more general and transparent manner is implemented on some virtual machine monitors. For example, KVM provides KSM(Kernel Samepage Merging) which merges common pages across multiple VMs (and also common pages within a VM). However, such method requires heavy scanning of memory and thus usually has long intervals between scans. It would be interesting to investigate if such scheme works well with our system.

Randomized Control Flow

Lastly we mention the work Branch-on-Random[16] which, although not a security related work, gave us an inspiration of our idea. It proposes a unique conditional branch instruction, where the condition is *probability*, i.e., decision to take the branch or not is determined probabilistically at the moment of branch instruction execution. It is similar to our approach in the sense that the control flow proceeds in random manner, though in our work the destination is randomly determined before execution.

7. CONCLUSION

In this paper we presented our novel idea for defending against control-flow hijacking attacks. This problem has been challenged by many research works in the past, but what distinguishes our work from the majority of them is the focus on making the approach satisfy not only security property, but also compatibility properties such as not requiring recompilation, code analysis, or preserving integrity of execution. This focus is derived from the insight that ASLR, which is the most widespread defense, satisfies the latter property sacrificing the former property.

The main purpose of this paper is the proposal of the overview and design of this approach, though we also implemented a prototype implementation on Pin to perform a brief evaluation on of its feasibility. We conclude that this approach is expected to be feasible, while for perfect compatibility it may require modifying the compiler, and such changes should be much simpler than other proposed methods. This work is expecting many further refinements and evaluation to be done in the future, which includes optimizations for improving runtime performance, compatibility improvements through crash analysis towards full support for dynamically loaded images. It would also be an interesting future work to investigate the advantages and requirements for implementing it on hardware, or application to Linux.

8. REFERENCES

[1] MSDN Library, “Windows ISV Software Security Defenses," http://msdn.microsoft.com/en-us/library/bb430720.aspx

[2] Android Developers, “Android 4.0 Platform Highlights," http://developer.android.com/sdk/android-4.0-highlights.html

[3] G. S. Kc, Angelos D. Keromytis, and V. Prevelakis, “Countering code-injection attacks with instruction-set randomization," in Proc. of the 10th ACM conference on Computer and communications security (CCS ’03), 2003.

[4] c0ntex, “Bypassing non-executable-stack during exploitation using return-to-libc", http://www.infosecwriters.com/text_resources/pdf/return-to-libc.pdf

[5] H. Shacham, “The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in Proc. of the 14th ACM Conference on Computer and Communications Security (CCS ’07), 2007.

[6] Phrack Magazine, “Bypassing PaX ASLR protection", http://www.phrack.com/issues.html?issue=59&id=9/

[7] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," Programming Language Design and Implementation (PLDI ’05), 2005.

[8] M. Chew and D. Song,“Mitigating buffer overflows by operating system randomization," Technical Report CMU-CS-02-197, Carnegie Mellon University, Dec. 2002.

[9] S. Forrest, A. Somayaji, and D. H. Ackley, “Building diverse computer systems," Workshop on Hot Topics in Operating Systems, 1997.

[10] D. Williams et al. “Security through Diversity: Leveraging Virtual Machine Technology," IEEE Security and Privacy 7, 2009.

[11] J. E. Just and M. Cornwell. “Review and analysis of synthetic diversity for breaking monocultures," in Proc. of the 2004 ACM workshop on Rapid malcode (WORM ’04), 2004.

[12] C. Wang, “Protection of software-based survivability schemes", Dependable Systems and Networks, 2001.

[13] M. Franz, “E unibus pluram: massive-scale software diversity as a defense mechanism," in Proc. of the 2010 workshop on New security paradigms (NSPW ’10), 2010.

[14] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh, “On the effectiveness of address-space randomization," in Proc. of the 11th ACM conference on Computer and communications security (CCS ’04), 2004.

[15] P. Philippaerts, Y. Younan, S. Muylle, F. Piessens, S. Lachmund, and T. Walter, “Code Pointer Masking: Hardening Applications against Code Injection Attacks," in Proc. of the 8th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA ’11), 2011.

[16] E. Lee and C. Zilles, “Branch-on-random," in Proc. of the 6th annual IEEE/ACM international symposium on Code generation and optimization (CGO ’08), 2008.

[17] C. Cowan, C. Pu, D. Maier, H. Hinton, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, “StackGuard: Automatic detection and prevention of buffer-overflow attacks," in Proc. of the 7th USENIX Security Symposium(Security ’98), 1998.

[18] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, “Pointguard: protecting pointers from buffer overflow vulnerabilities," in Proc. of the 12th USENIX Security Symposium(Security ’03), 2003.

[19] “Bzip2 for Windows," http://gnuwin32.sourceforge.net/packages/bzip2.htm

[20] MSDN, “Dynamic-Link Librariesk," http://msdn.microsoft.com/en-us/library/windows/desktop/ms682589%28v=vs.85%29.aspx

[21] Dionysus Blazakis, “Interpreter Exploitation: Pointer Inference and JIT Spraying," in BlackHat DC, 2010.

[22] A. Skaletsky, T. Devor, N. Chachmon, R. Cohn, K. Hazelwood, V. Vladimirov, and M. Bach, “Dynamic program analysis of Microsoft Windows applications," in Proc. of IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS ’10), 2010.

[23] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis, “Smashing the Gadgets: Hindering Return-Oriented Programming Using In-Place Code Randomization," in the Proc. of the 33rd IEEE Symposium on Security & Privacy (S&P ’12), 2012.

[24] L. Davi, A.-R. Sadeghi, and M. Winandy, “ROPdefender: A practical protection tool to protect against return-oriented programming," in Proc. of the 6th Symposium on Information, Computer and Communications Security (ASIACCS ’11), 2011.

[25] metasploit Exploit DB, “Windows ANI LoadAniIcon() Chunk Size Stack Buffer Overflow (SMTP)", http://www.metasploit.com/modules/exploit/windows/email/ms07_017_ani_loadimage_chunksize

[26] Bugzilla, “Mandatory ASLR on Windows for binary components," https://bugzilla.mozilla.org/show_bug.cgi?id=728429

[27] Microsoft TechNet Blogs, “Introducing EMET v3," http://blogs.technet.com/b/srd/archive/2012/05/15/introducing-emet-v3.aspx

[28] PaX Team, “address space layout randomization," http://pax.grsecurity.net/docs/aslr.txt

[29] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, “Return-oriented programming without returns," in Proc. of the 17th ACM conference on Computer and communications security (CCS ’10), Oct 2010.

[30] PaX Team, “vma mirroring," http://pax.grsecurity.net/docs/vmmirror.txt

[31] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie, “DROP: Detecting return-oriented programming malicious code," in Proc. of the 5th International Conference on Information Systems Security (ICISS ’09), 2009.

[32] B. Salamat, A. Gal, and M. Franz, “Reverse stack execution in a multi-variant execution environment," in Workshop on Compiler and Architectural Techniques for Application Reliability and Security(CATARS), 2008.

[33] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, “Enhanced operating system security through efficient and fine-grained address space randomization," in Proc. of the 21st USENIX Security Symposium(Security ’12), 2012.

[34] R. Wartell, V. Mohan, K. W. Hamlen, Z. Lin, “Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code," in Proc. of the 19th ACM conference on Computer and communications security (CCS ’12), 2012.

Distributed Application Tamper Detection Via Continuous Software Updates

Christian Collberg
University of Arizona
collberg@cs.arizona.edu

Sam Martin
University of Arizona
smartin336@cs.arizona.edu

Jonathan Myers
University of Arizona
jamyers@cs.arizona.edu

Jasvir Nagra
Google Inc.
jasvir@nagras.com

ABSTRACT

We present a new general technique for protecting clients in distributed systems against *Remote Man-at-the-end* (R-MATE) attacks. Such attacks occur in settings where an adversary has physical access to an untrusted client device and can obtain an advantage from tampering with the hardware itself or the software it contains.

In our system, the trusted server overwhelms the analytical abilities of the untrusted client by continuously and automatically generating and pushing to him diverse client code variants. The diversity subsystem employs a set of primitive code transformations that provide an ever-changing attack target for the adversary, making tampering difficult without this being detected by the server.

Categories and Subject Descriptors

K.6.5 [**Security and Protection**]

General Terms

Renewability, Defense-in-depth, Diversity, Obfuscation, Tamperproofing

Keywords

Distributed Systems, Security, Software Protection

1. INTRODUCTION

Man-at-the-end (MATE) attacks occur in settings where an adversary has physical access to a device and compromises it by tampering with its hardware or software. *Remote* man-at-the-end (R-MATE) attacks occur in distributed systems where *untrusted clients* are in frequent communication with *trusted servers* over a network, and malicious users can get an advantage by compromising an untrusted device.

To illustrate the ubiquity of R-MATE vulnerabilities, consider the following four scenarios. First, in the *Advanced Metering Infrastructure* (AMI) for controlling the electrical power grid, networked devices ("*smart meters*") are installed at individual households to allow two-way communication with control servers of the utility company. In an R-MATE attack against the AMI, a malicious consumer tampers with the meter to emulate an imminent blackout, or to trick a control server to send disconnect commands to other customers [7, 22]. Second, massive multiplayer online games are susceptible to R-MATE attacks since a malicious player who tampers with the game client can get an advantage over other players [17]. Third, wireless sensors are often deployed in unsecured environments (such as theaters of war) where they are vulnerable to tampering attempts. A compromised sensor could be forced into supplying the wrong observations to a base station, causing real-world damage. Finally, while electronic health records (EHR) are typically protected by encryption while stored in databases and in transit to doctors' offices, they are vulnerable to R-MATE attack if an individual doctor's client machine is compromised.

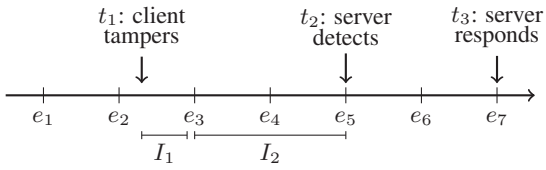
ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

1.1 Overview

In each of the scenarios above, the adversary's goal is to tamper with the client code and data under his control. The trusted server's goal is to *detect* any such integrity violations, after which countermeasures (such as severing connections, legal remedies, etc.) can be launched.

Security mechanisms. In this paper we present a system that achieves protection against R-MATE attacks through the extensive use of code diversity and continuous code replacement. In our system, the trusted server continuously and automatically generates diverse variants of client code, pushes these code updates to the untrusted clients, and installs them as the client is running. The intention is to force the client to constantly analyze and re-analyze incoming code variants, thereby overwhelming his analytical abilities, and making it difficult for him to tamper with the continuously changing code without this being detected by the trusted server.

Limitations. Our system specifically targets distributed applications which have frequent client-server communication, since client tampering can only be detected at client-server interaction events. Furthermore, while our use of code diversity can *delay* an attack, it cannot completely *prevent* it. Our goal is therefore the rapid *detection* of attacks; applications which need to completely prevent any tampering of client code, for even the shortest length of time, are not suitable targets for our system. To see this, consider the following timeline in the history of a distributed application running under our system:



The e_i's are *interaction events*, points in time when clients communicate with servers either to exchange application data or to perform code updates. At time t_1 the client tampers with the code under his control. Until the next interaction event, during interval I_1, the client runs autonomously, and the server cannot detect the attack. At time t_2, after an interval I_2 consisting of a few interaction events, the client's tampering has caused it to display anomalous behavior, perhaps through the use of an outdated communication protocol, and the server detects this. At time t_3, finally, the server issues a response, perhaps by shutting down future interactions. Programs in which the client does not need to frequently communicate with the server are not suitable to our system.

Adversarial model. We place very few limiting assumptions on the adversary's abilities. First of all, we expect that, through reverse engineering, he has achieved a complete understanding of our system. We also expect him to have at his disposal all the common static and dynamic analysis techniques that would be helpful in analyzing and tampering with the code under his control, including debuggers, tracers, disassemblers, decompilers, control- and data-flow analyses, code slicers, etc. The level of protection against tampering that our system affords thus does not rely on security-by-obscurity but rather on the server's ability to deliver to the client a steady stream of random code mutations, at a rate that exceeds the client's ability to reverse engineer. To ensure this, all of our transformations are randomized, and many are based on hard static analysis problems, such as the need to perform pointer analysis.

Security vs. performance trade-offs. Some of our code transformations incur much overhead on the client side (we can, for example, transform a function by applying multiple levels of randomized interpretation), and some incur overhead on the server side (generating streams of code variants for multiple clients). Our system is therefore completely configurable, allowing the server to trade-off between its ability to detect client tampering, client performance, and server performance.

2. BACKGROUND AND RELATED WORK

Fred Cohen was the first to suggest that code transformations could be used to create a diverse set of programs which would be less vulnerable to attack [8]. The intuition is that if every installation of, say, an operating system, is different, adversaries (human intruders as well as automated malware) will find it more difficult to reuse an attack that was successful against one variant when attacking another. Fritz Hohl may have been the first to suggest (in the context of protecting mobile agents) *time-limited* protection against MATE attacks [18]. The intuition is that an adversary who is in complete control over his attack target may still be kept at bay if his window of opportunity is sufficiently short; i.e. if the time needed to analyze and tamper with the target is strictly longer than the time the target is of value to the attacker. In this paper we combine these two ideas: we protect vulnerable code running on an untrusted client in a distributed system by diversifying the code and, by continuously and automatically replacing that code, we ensure that the client has a short time window to analyze and tamper with it.

Code diversity. Since Cohen's original paper, several systems have been developed to randomize various aspects of computer systems to protect against attacks: opcodes can be randomized and then executed under emulation/interpretation [5, 1]; modules, stacks, and heaps can be placed in random location in memory (*Address Space Layout Randomization*, ASLR [31]); and code can be reordered and stack frame layouts can be randomized [15]. Other than ASLR, these ideas have not found their way into production systems. The reasons are many, but include unacceptable performance overhead and challenging software engineering problems, such as how one would approach software testing, error reporting, and perform updates on widely divergent programs.

Various forms of code randomization have also been used to protect programs against MATE attacks, i.e. to make programs more difficult to reverse engineer, tamper with, and redistribute illegally [9]. Common techniques include obfuscating code transformations that reorder code to make it difficult to analyze, tamperproofing transformations that reorder code at runtime to make it difficult to modify [4], and software watermarking techniques that embed unique identifiers in code, using compilers that randomize code-generation choices [2]. Such MATE protection techniques have mainly found applications in intellectual property protection of computer programs and digital media.

MATE attack models. The fundamental difference between MATE and other computer security scenarios is that a true MATE adversarial model cannot assume an unassailable *root-of-trust*. Because the adversary has physical access to the software and hardware of the device he tampers with, we have to assume that, given enough time, all defenses will be compromised. This is true even if the device is equipped with tamper-resistant hardware—experience has shown that such devices are eminently vulnerable to invasive as well as non-invasive attacks [3, 12]. This makes the problem of defending against MATE attacks significantly more challenging than for more traditional security scenarios.

In analogy with *Kerckhoffs's principles* for cryptography, in designing defenses against MATE attacks we must assume everything about our system is available to the attacker for study, including primitive code transformations, strategies for combining such transformations, and in fact, the source code of the entire system. What is *not* available to the attacker, however, and the only leverage we as defenders have, are the actual randomization seeds used in a particular run of the system. This means that the attacker cannot easily predict the *order* in which transformations are applied or the *location* in the code where they are applied.

R-MATE defense strategies. The Trusted Platform Module (TPM) chip has been suggested as a solution to the R-MATE problem, by ensuring that the untrusted client's software and hardware are approved and untampered [23]. This solution, however, has been shown to be impractical, requiring comprehensive white-lists of trusted software to be kept up-to-date at all times.

The CodeBender system [6] is the most closely related to our work, in that it uses code updates for security. Unlike our system, however, they rely on a single code transformation and require clients and servers to both be completely shut down and restarted for every update. For many distributed applications this is unacceptable.

Also related to the work presented here is the work of Stoyle et al. [29] who do run-time code updates, but in their case for servers, not clients, and not for security. Haeberlen et al. [16] also detect code tampering in distributed scenarios, but with a much more significant time delay until detection. The Pioneer system by Seshadri et al. [27] detects code tampering in a system where exact system specifications and latency of the network must be fully known, whereas our system works on general systems and networks such as the Internet.

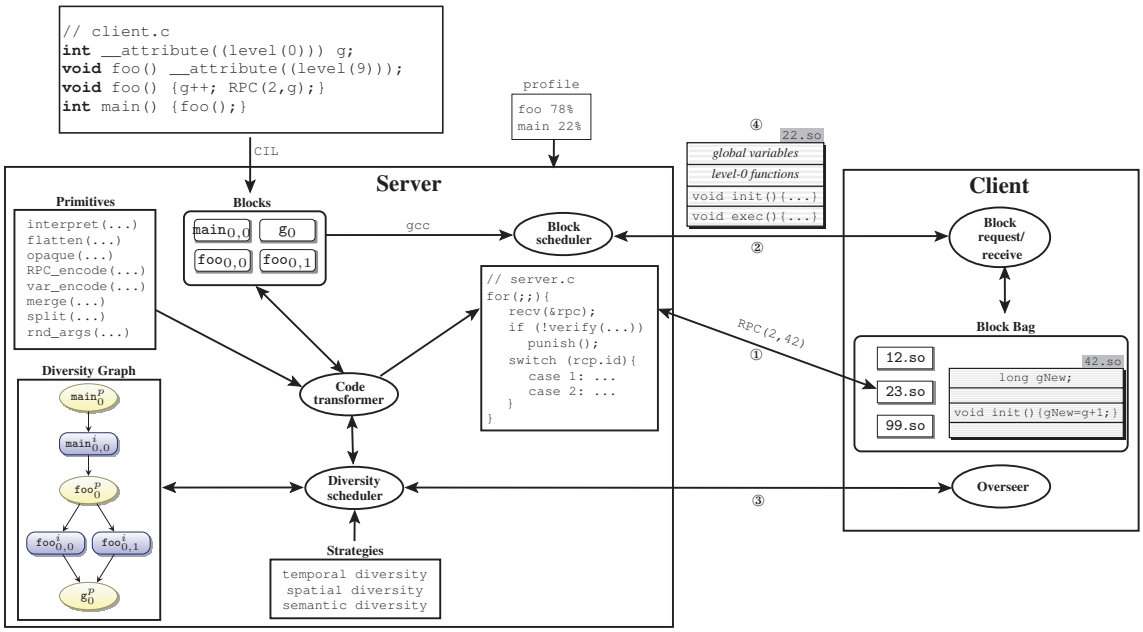


Figure 1: Overview of the diversity system.

The work by Scandariato et al. [24] also attempts to solve the issue of remote tamper-detection via software. They generate proofs that the code has been untampered, potentially by checksums. Their work doesn't make modifications to the code of the target application and obfucsations are not utilized. Falcarin et al. [14] break a program into pieces and send each over individually when the target requires them. The program is loaded into random memory locations, but no obfuscation is done at the function level.

3. CORE MECHANISMS

Our system (shown in Figure 1) consists of a trusted server and (one or more) untrusted clients. The input to the server is the code to be executed on the client side (`client.c`), the server code itself that services requests from the client (`server.c`), and profiling data for the client. The server contains a set of code and data *blocks* extracted from `client.c`, a set of *primitives* (block transformations), a set of *diversity strategies* for combining such primitives, and a *diversity graph* that expresses the relationships between every generated block variant. The system can easily be extended with new primitives.

The client holds (a subset of) the blocks that make up `client.c` in its *block bag*. The adversary's intent is to analyze and modify some of the blocks in the bag to meet his attack goal. The server's intent is to detect when it happens, and then to punish the client accordingly. The server's strategy is to accomplish this by making the block bag change continuously at runtime, forcing the client to invalidate old block variants and add new ones, thereby stressing his analytical ability.

Over the course of execution the client's block bag will be filled with a sequence of *block working sets*, $\langle w_0, w_1, \ldots \rangle$. Each $w_i = \{\ldots, \mathtt{f}_{j,k}, \ldots\}$ is a set of blocks which, taken together, forms a complete variant of the original `client.c`. A block $\mathtt{f}_{j,k}$ is a variant of a function f with *protocol* version number j and *implementation* version number k. At any one point in time, the client may only see a subset of the current working set, namely the set of blocks it actually needs for execution. Moving from working set w_i to w_{i+1} normally only changes a subset of the blocks, i.e. it is the result of applying a few primitive code transformations to a few of the blocks in w_i. In Figure 1, for example, it appears that an initial working set was $w_0 = \{\mathtt{main}_{0,0}, \mathtt{foo}_{0,0}, \mathtt{g}_0\}$ followed by a transformation of `foo`, yielding a second generation working set $w_1 = \{\mathtt{main}_{0,0}, \mathtt{foo}_{0,1}, \mathtt{g}_0\}$.

There are three concurrent tasks occurring at runtime. The first task (① in Figure 1), is the "normal" client-server interaction that would occur in an unprotected distributed application, where the client executes code out if its block bag and makes remote procedure calls (RPC) to the server whenever this is needed in order to make progress. The code in `server.c` continuously listens for and services such requests. In the second task (②), the server's *block scheduler* supplies the client with blocks it requests or pushes blocks to the client that it predicts the client may need. In the third task (③), finally, the server's *diversity scheduler* informs the client's *overseer* process that a new update cycle is about to begin, the client returns its current state (blocks on the current call stack, which we will refer to as the *active set*) to the server and suspends itself. The server then generates new block variants by selecting appropriate primitives and strategies, invokes the *code transformer* to apply these transformations, and, once done, wakes up the client to inform it of invalidated blocks.

The code transformer and the primitives are built on top of the CIL [20] system for C code transformations. CIL's facilities for static analysis are needed to implement the primitive code transformations. At system startup time, `client.c` is parsed by CIL and each function and global variable (also called *blocks* since they are one-to-one with the blocks in the client's block bag) is extracted and kept in memory in CIL's intermediate code format. Blocks are compiled into dynamically linked libraries (`.so` files) which the block scheduler transfers to the client for execution. Each block (④ in Figure 1) contains four sections, each of which can be empty: zero or more global variables, zero or more unprotected (level-0) functions, an initialization routine (`init`) which is executed when a block first arrives in the client's block bag, and a main routine `exec` which is executed when a function gets called.

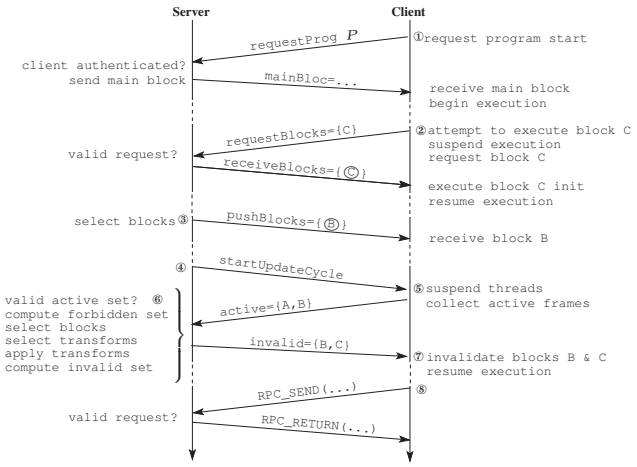


Figure 2: Client-server communication timeline.

3.1 Client-Server Protocol

To illustrate the interaction between a server and a client, consider the timeline in Figure 2. Execution begins at ① where, after proper authentication, the client receives the initial block, containing global variables, level-0 functions, and `main` itself. Subsequently, the client can either explicitly request a block (②) that it needs to execute, or receive blocks in the background (③) that the server predicts that it will need. Whenever a block request is received by the server it performs a validation step: given what is known of the client's current state (information gathered from active sets received during the update cycle, the sequence of block requests, and remote procedure calls the client has made), the server determines whether the client is in compliance.

At ④, the server initiates an update cycle. At ⑤, the client suspends execution and responds with the active set of blocks (the current call stack). At ⑥, the server checks that the active set forms a valid sequence of function calls, computes the forbidden set (blocks which cannot be obfuscated), selects blocks to be transformed and the transformations to be applied, applies the transformations and sends the invalid set (out-of-date blocks which the client should no longer execute) to the client. At ⑦, the client resumes execution, concluding the update.

For every incoming remote procedure call (⑧), the server also verifies the validity of RPC numbers and argument values.

3.2 Transformation Primitives

A *primitive* in our system is a code transformation that transforms a function or variable into a different form. The primitives serve three goals: to add confusion to the client code thereby taxing the adversary's analytical abilities (obfuscation); to make it more difficult for the adversary to modify the client code (tamperproofing); and to make it easier for the trusted server to detect any tampering of the client code (tamper-detection). These goals are not independent: adding confusion to client code makes static and dynamic analysis harder, and this in turn makes the adversary's goal of modifying the client code while maintaining correctness more difficult.

3.2.1 Preserving Protocols

Transformations that only add confusion are, by themselves, not sufficient for our purposes. Consider, for example, a situation where, in accordance with Kerckhoffs's principles, the adversary has full access to our system and, through analysis, has realized that our primitives only add non-functional code to functions while otherwise preserving their behavior. He can then simply reverse engineer a function the first time he sees it, tamper with it at will, and then ignore any further code updates! Therefore, some of our primitives generate variants that are compatible with each other, while some do not. Our goal is for the system to add enough confusion such that the adversary cannot easily determine whether a code update pushed to it can be safely ignored or not. If our primitives are able to satisfy this goal, we will force the adversary to expend resources on fully analyzing (automatically, or ideally, manually) each new block that gets sent to him. Then, providing security against R-MATE attacks becomes a question of the system being able to sustain a block replacement rate that is high enough to overtax the adversary's analytical abilities, without causing too much computational overhead.

Thus, some of our transformations are *protocol-preserving*. This means they only change the body of a function, but the interface through which the function is called is preserved. Other transformations are *non-protocol-preserving*, meaning any code that calls a transformed function must be updated to use the new interface. Similarly, a primitive that modifies the encoding of a global variable does not maintain protocol compatibility—any function which references that variable needs to be updated to use the new encoding. We get the following definitions:

DEFINITION 1 (PROTOCOL COMPATIBILITY). *T is known as a* protocol-preserving *transformation if p is a function or a remote procedure and $T(p)$ preserves p's external behavior, its signature and the encoding of arguments and return values. T is a protocol-preserving variable transformation if p is a variable and $T(p)$ preserves p's type and encoding.* □

Block variants are numbered with a protocol and implementation version number: $\texttt{block}_{\text{prot,impl}}$. Two blocks with identical protocol numbers (such as $\texttt{foo}_{0,0}$ and $\texttt{foo}_{0,1}$ in Figure 1) are protocol compatible.

3.2.2 Diversifying Primitives

Some of the primitives described here are adapted from obfuscating code transformations found in the literature. The adaptations *parameterize* the primitives such that a given transformation can generate multiple different variants from the same function. Transformations take a random number seed as input and use a PRNG to randomize choices they make.

It should be noted that all our transformations can be combined *ad infinitum*: a function can first be flattened and then turned into an interpreter, a second layer of interpretation can be added (using a different dispatch method and instruction set), etc. This is important for transformations which in themselves do not have a whole lot of opportunities for diversification, but which can be combined to generate a potentially infinite sequence of variants. It is the duty of the strategies in the diversification scheduler (see Section 3.4) to decide when and how to combine primitives.

3.2.3 Protocol-Preserving Primitives

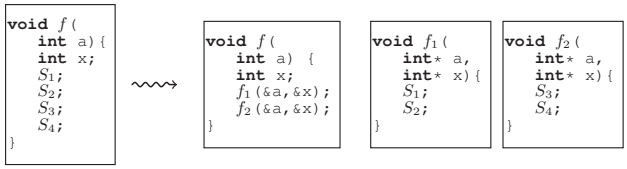
Our current implementation supports four protocol-preserving transformations which only modify the body of functions, not the way in which they are invoked.

The $\textbf{interpret}(f, \text{seed})$ transformation turns a function f into a specialized interpreter for f. The seed is used to select a random method of instruction dispatch (currently supported are call, switch, direct and indirect threading [13]), to select a fraction of instruction pairs to merge into superoperators [21], to randomize opcode

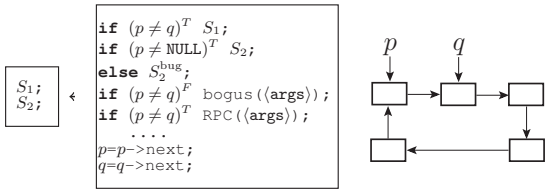
assignment, and to select the mix of addressing modes (register and stack arguments).

The **flatten**(f, seed) transformation removes nested control flow from a function f by making every basic block a case in a switch statement nested inside an infinite loop [30]. The seed is used to randomize the order of the basic blocks inside the switch statement.

The **split**(f, seed) transformation converts a function f into two functions f_1 and f_2, now called from f. The seed randomizes the point in f where the split is made. f passes along its local environment (formal arguments and local variables) to f_1 and f_2:



The **opaque**(f, seed) transformation inserts non-functional code protected by an opaque predicate [10]. This code can range from code that causes crashes, code that invalidates client data, or calls to fake functions which the server knows should never be requested. The seed randomizes the point in f where the insertion is made, and also the type of transformation. This example shows four of the kinds of transformations currently supported (S_2^{bug} is an obfuscated version of S_2 into which a bug has been inserted, `bogus` is a randomly selected function, and ⟨**args**⟩ a list of random expressions):



Here, P^T represents an opaquely true predicate, i.e. a boolean expression that is difficult for an adversary to analyze. In our current implementation, opaque predicates are manufactured by creating and dynamically modifying linked structures. In the example above, p and q are pointers being moved around in a circular linked list subject to the invariant $p \neq q$.

3.2.4 Non-Protocol-Preserving Primitives

Our system currently supports four primitives that do not preserve protocol compatibility, namely **rnd_args**, **RPC_encode**, **merge**, and **var_encode**.

The **merge**(f_1, f_2, seed) transformation combines two functions $f_1(\text{args}_1)$ and $f_2(\text{args}_2)$ into a new function with a signature of $f_{1,2}(\text{args}_1||\text{args}_2, \text{sel})$, in which sel is used at the call site to distinguish between calls to the two functions.

The **rnd_args**(f, seed) primitive randomly reorders f's formal parameters and adds extra, bogus, formals.

We have extended C with a remote procedure call mechanism that allows for simple client-server communication. Remote procedure calls are identified by number and carries with them a sequence of scalar data items and return a scalar result. The transformation of **RPC_encode**(n, seed) assigns a new random encoding of the n'th remote procedure call RPC(n, args) by assigning it a new random RPC number, randomly reordering its arguments, and inserting bogus arguments. This is an important transformation since, should an adversary choose to ignore block updates, he may inadvertently execute an invalid block containing an RPC with an obsolete encoding, thereby alerting the server of the tampering.

The **var_encode**(g, seed) transformation selects a random new encoding for a variable g [11]. As a trivial example, a global integer variable `x` could be replaced with a long variable `x_new` whose encoding is `x+1`. Any reference throughout the program to `x` in the original code has to be updated in order to preserve correctness. When the **var_encode**$(\mathtt{x}, \text{seed})$ transformation is selected, not only do all blocks containing `x` need to be updated, but we also need to allocate the new variable on the client side and initialize its value to the current value of `x`, plus one. This is done by sending the client an `init` block like `42.so` in Figure 1.

The **var_encode** transformation is important in that an adversary who attempts to use an obsolete block may inadvertently use an old variable encoding. This could lead to client failure or, when the server is sent the old variable's value in an RPC, detection by the server.

3.3 Diversity Graph

Our transformations generate a complex set of relationships between blocks: function signatures change, RPC arguments are permuted, global variables change in type and encoding, and so on, and these changes, in turn, can force changes to other blocks. While this complexity is good for creating confusion on the client side, it unfortunately also results in generations of block working sets with varying levels of compatibility.

To allow us to reason intelligently about the relationships between various block variants, we use an abstraction we call a **diversity graph** representing dependencies between blocks and protocols. The scheduler uses the diversity graph to determine which blocks to transform next and how a transformation applied to one block will force updates to other blocks.

In the initial program, the diversity graph is nearly identical to a conventional call graph, with the addition of nodes for global variables and edges between a function and the global variables it references. In addition to the nodes that represent function *implementations*, we also add nodes to represent function *protocols*. As transformations are applied the graph will grow with nodes for all new variable/function variants.

DEFINITION 2 (DIVERSITY GRAPH). *A diversity graph G for a program P consists of the following* protocol *and* implementation *nodes, where m is a protocol and n an implementation number:*

- *an implementation node* $\mathtt{f}^i_{m,n}$ *for each function variant;*
- *a protocol node* $\mathtt{f}^p_m$ *for each function protocol;*
- *a protocol node* $\mathtt{RPC(n)}^p_m$ *for each RPC protocol;*
- *a protocol node* $\mathtt{v}^p_m$ *for each protocol of a variable* v.

G contains the following directed edges:

- $\mathtt{f}^p_m \rightarrow \mathtt{f}^i_{m,n}$ *from a function protocol node to its variant implementations;*
- $\mathtt{f}^i_{m,n} \rightarrow \mathtt{g}^p_s$ *if* $\mathtt{f}^i_{m,n}$ *calls one of* $\mathtt{g}^p_s$*'s implementations;*
- $\mathtt{f}^i_{m,n} \rightarrow \mathtt{RPC(v)}^p_s$ *if* $\mathtt{f}^i_{m,n}$ *makes remote procedure call number* v.
- $\mathtt{f}^i_{m,n} \rightarrow \mathtt{v}^p_s$ *if* $\mathtt{f}^i_{m,n}$ *references global variable* $\mathtt{v}^p_s$. □

Forward update cycles. Figure 3 shows the effect on the diversity graph from a sequence of *forward update cycles*, where each new graph is the result of applying one primitive. Figure 3 (a) shows the initial graph for a program consisting of a global variable `g` and two functions `main` and `foo`. Note that each function results in a single protocol node and a single implementation node. A global variable is represented by a single protocol node. Figure 3 (b) shows the graph after an application of the protocol-preserving primitive **flatten**, which adds a new implementation

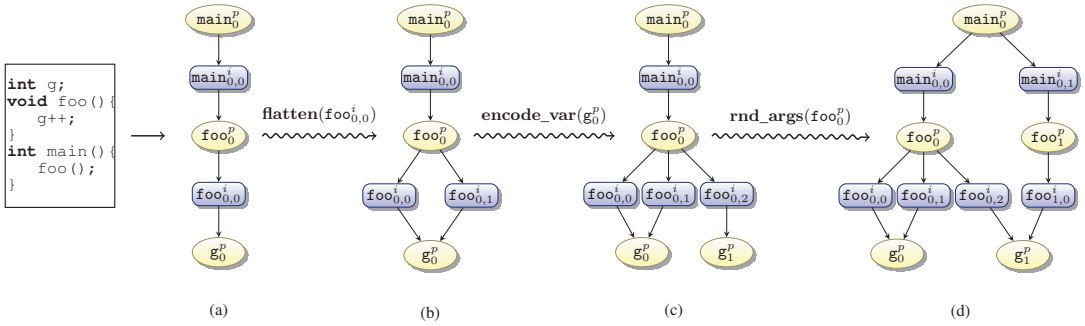


Figure 3: Diversity graphs. Protocol nodes are ellipses and implementation nodes rectangles.

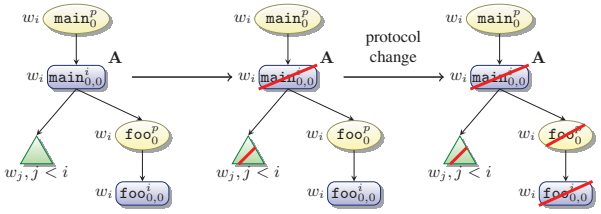


Figure 4: Computing forbidden sets for forward operations. Active blocks are marked with an A, w_i marks the generation a block belongs to, and forbidden blocks are crossed out.

node $\mathtt{foo}^i_{0,1}$. Next, we obfuscate $\mathtt{g}^p_0$ using the **encode_var** primitive, which generates a protocol-incompatible variant $\mathtt{g}^p_1$. We are forced to furthermore generate a new variant $\mathtt{foo}^i_{0,2}$ of $\mathtt{foo}^i_{0,1}$ compatible with $\mathtt{g}^p_1$. In Figure 3 (d), finally, $\mathtt{foo}^i_{1,0}$ is created by applying **rnd_args** to $\mathtt{foo}^p_0$, resulting in a new protocol node $\mathtt{foo}^p_1$.

Rollback update cycles. For performance reasons it is not always possible to transform a block working set w_i into a new generation w_{i+1} by simply applying obfuscating primitives. Every such transformation incurs overhead and compounding transformations would eventually lead to unacceptable client performance. The diversity scheduler supports a *rollback* operation, in which working set w_{i+1} is the result of replacing *some of* the blocks in w_i with variants from previous generations. Since our primitives are parameterized, this will allow the system to first roll back a few blocks to previous variants, then roll forward by applying new transformations, and to do so without incurring any extra performance penalty.

3.4 Diversity Strategies

It is well known that software protection techniques (such as the obfuscating primitives in Section 3.2) deployed in the field have a limited survival time. Any static target will fall to a motivated adversary equipped with standard reverse engineering tools such as debuggers, tracers, disassemblers, and decompilers. We therefore make use of *strategies*, means of combining primitive transformations that gives the adversary not a static, but a constantly changing, attack target. We identify three strategies, based on the notion of diversity, essential to making effective use of software protection:

DEFINITION 3 (DIVERSITY). *A program p is* temporally diverse *if it is delivered to the user, over time, as an infinite and non-repeating sequence of variants $\langle v_0, v_1, \ldots \rangle$ where $v_i \neq v_j$ if $i \neq j$. A program p is* spatially diverse *if a variant v is constructed by compounding multiple layers of interchangeable primitive transformations. A program p is* semantically diverse *if two variants v_i and v_j of p cannot be used interchangeably.* □

Temporal diversity is sometimes known as *renewability* or *software aging* [19], and spatial diversity as *defense-in-depth*.

The diversity scheduler (see Figure 1) decides on an appropriate sequence of strategies to employ in order to best protect the client program from tampering. It employs temporal diversity by producing generations of block variant working sets, spatial diversity by compounding protocol-preserving primitives from Section 3.2.3, and semantic diversity by applying the non-protocol-preserving primitives from Section 3.2.4.

The input to the scheduler is the current working set w_i, the current diversity graph G_i, the performance profile of the client code, the security requirements of different parts of the code (the `level` attributes provided as program annotations), the set of primitives and their effect on diversity and performance, and the *active* set (functions currently on the client's call stack). The output of the scheduling operation is a new working set w_{i+i}, the *kill set* of blocks that need to be invalidated on the client side, and an updated diversity graph G_{i+1}.

3.4.1 The Diversity Scheduler

The scheduler can choose between *forward* and *rollback* updates. In a forward update it applies primitive transformations to the blocks in the current working set, resulting in a new working set with more confusion but also more performance overhead. In a rollback update, one or more of the blocks in the current working set are made to revert back to previous variants.

Depending on the state that the client is currently in, not all blocks can be transformed by all primitives, nor can all blocks be reverted to previous variants. Fundamentally, the reason is that our system is designed not to replace running code, i.e. functions currently on the activation stack. We will next examine the restrictions necessary in the case of forward cycles.

3.4.2 Restrictions on Forward Update Cycles

To see the restrictions on which transformations can be applied to which blocks in case of a forward update cycle, consider the example in Figure 4 (a), where $\mathtt{main}^i_{0,0}$ is on the stack. Since it would provide no further diversity to transform blocks not in the current generation, any block in $w_j, j < i$ is forbidden. Active blocks (here, only $\mathtt{main}^i_{0,0}$) are also forbidden since in our system we cannot replace running code. Furthermore, a block called directly by an active block is forbidden for non-protocol-preserving transformations. To see why, consider $\mathtt{foo}^i_{0,0}$, above. If we were to change $\mathtt{foo}^i_{0,0}$'s protocol, the call site in $\mathtt{main}^i_{0,0}$ would have to be updated, and this is not possible since this block is currently active.

Thus, to perform a forward transformation, the scheduler computes forbidden nodes in the diversity graph, selects a permissible block B and a transformation T, performs the transformation, updates the diversity graph, and sends the kill set to the client. If there is more than one possible candidate block and transformation, we choose heuristically to maximize diversity and confusion while minimizing additional overhead.

3.5 Linking

During execution blocks will continuously come and go in the client's block bag. Therefore, it is not possible to refer to code blocks directly, and we are forced to add a level of indirection. We thus keep an array of function pointers `funPtrArr` indexed by block numbers. A function call `bar(42)`, where `bar` is block number 22, turns into the following code, where the function named `_loadBlock(blockNo)` loads a missing block over the network and fills in `funPtrArr[blockNo]` with the new address:

```
float (*tmp)(int) = funPtrArr[22];
if (tmp == NULL) {tmp = _loadBlock(22);}
(*(float(*)(int))tmp(42);
```

4. SECURITY ANALYSIS

In this section we evaluate the security generated by our system. In Section 4.1 we first discuss the server's ability to detect tampering during interaction-events with the client. In Section 4.2 we enumerate the methods of attack available to the client and how each of these attacks can be blocked. In Section 4.3 we describe the results of attacks we implemented and executed ourselves. In Section 4.4, finally, as a measure of the difficulty of reusing analyses (the most serious attack described in Section 4.2) we measure the diversity created by our various transformations.

4.1 Attack Detection

There are several ways in which the server can check that the client is playing by the rules and ways in which it can punish him if he does not. Referring back to Figure 1, we see that there are three client-server interaction events which give the server the opportunity to detect foul play.

In Figure 1 ②, the block scheduler receives a block request from the client, at which time it will verify that the block belongs to the client's current working set. If it does not, this means that the client has refused a previous block update and, as a result, is now requesting obsolete blocks belonging to a previous working set. If fake function calls have been added into the program, a request for a fake function (which has been inserted by means of the **opaque** primitive) will also alert the server that tampering has occurred.

In Figure 1 ③, the server initiates an update cycle and the client responds with its active set, a list of the functions on its current call stack (or call stacks, in case of a multi-threaded program). Clearly, it is in the client's best interest to lie about the active set: if he can convince us that the entire working set is active, i.e. that every function in the program is currently on the call stack, he will prevent us from transforming any blocks! The block scheduler can detect such degenerate cases by using the call graph portion of the diversity graph to verify that the active sets represent realizable call sequences.

In Figure 1 ①, `server.c` verifies that a remote procedure call from the client uses a current protocol. This includes checking that the RPC number and the type and order of arguments is current, as well as checking that all arguments to the call are within their specified ranges. Since the **RPC_enc** primitive will occasionally be invoked to randomize the protocol, this is our main way of verifying that the client is executing blocks from an untampered and current working set.

We can also add application specific checks to `server.c`. For example, we may know that certain sequences of remote procedure calls are illegal, or we may be able to keep track of enough of the client's current local state to be able to detect calls that could never happen at the current point in the execution.

4.2 Enumeration of the Attack Space

The attack tree in Figure 5 shows the attack space available to a malicious user. The root of the tree represents the attacker's goal, namely to tamper with a particular *asset* in the program without being detected. An asset could be a security check, code that updates a particular global variable, the integrity of a control-flow path, global data, etc. An effective attack proceeds in three steps: successfully find the *asset blocks* $\mathcal{A} = \{A_1, \ldots, A_n\}$, the set of blocks that need to be modified among the blocks in the block bag (node **1**); successfully tamper with these blocks (node **2**), and, finally, avoid detection by the server (node **3**). Once the asset blocks have been located, the adversary is free to modify them to reach his attack goals.

Figure 5 shows four ways for the attacker to avoid the server detecting the modification to $\mathcal{A}$, ordered from easiest to hardest to carry out. First (node **3.1**), if the attacker is lucky the asset blocks are badly protected, i.e. not connected to the rest of the program through use of global variables, calls to other functions, or calls to RPCs, in any meaningful way. Then he can modify $\mathcal{A}$ without the risk of detection.

Node **3.2** shows that if, during every update cycle, the client can convince the server that the blocks in $\mathcal{A}$ are on the $active$ set, then the server will never be able to update $\mathcal{A}$. The active set must still be plausible, requiring the client to analyze the blocks, build the call graph, and report an $active$ set that does not arouse suspicion.

Node **3.3** shows that the client can trick the server into only making trivial changes to $\mathcal{A}$, allowing him to simply ignore updates. For this attack to work the client reports an $active$ set that contains blocks that reference the RPCs and variables that $\mathcal{A}$ also references. As a result, those RPCs and variables cannot be modified, which prevents the server from performing major changes to $\mathcal{A}$.

Finally, node **3.4** shows an attack where the adversary finds ancestor blocks $N_0, \ldots N_{k-1}$ of a new block variant N_k, extracts any new variable or RPC encodings, and patches $\mathcal{A}$ to use these new encodings. Thus, $\mathcal{A}$ remains tampered but conforms to any new protocols.

4.2.1 Countermeasures

The primitives in Section 3.2 have been designed to counter the attacks above.

Nodes **1.2.1, 3.1, 3.2.1.1, 3.3.1.1**, and **3.4.1** in Figure 5 show that block analysis is an integral part of every attack. Depending on the information needed to carry out the attack, this will involve various forms of static and/or dynamic analyses, such as disassembly, decompilation, control-flow analysis, data-flow analysis, slicing, etc. We use the primitives, especially the protocol-preserving ones in Section 3.2.3, to make block analysis more difficult. Any reverse engineering effort will eventually succeed, but because blocks are constantly updated it is sufficient for the transformations to slow down the attacks between updates.

If the asset is contained in one or more orphan blocks (node **3.1**), such as the unlikely situation where the asset is a function containing a single `printf("hello world")` statement, the adversary can modify it at will. In such situations the **opaque** primitive will allow us to connect the orphan block to the rest of the program

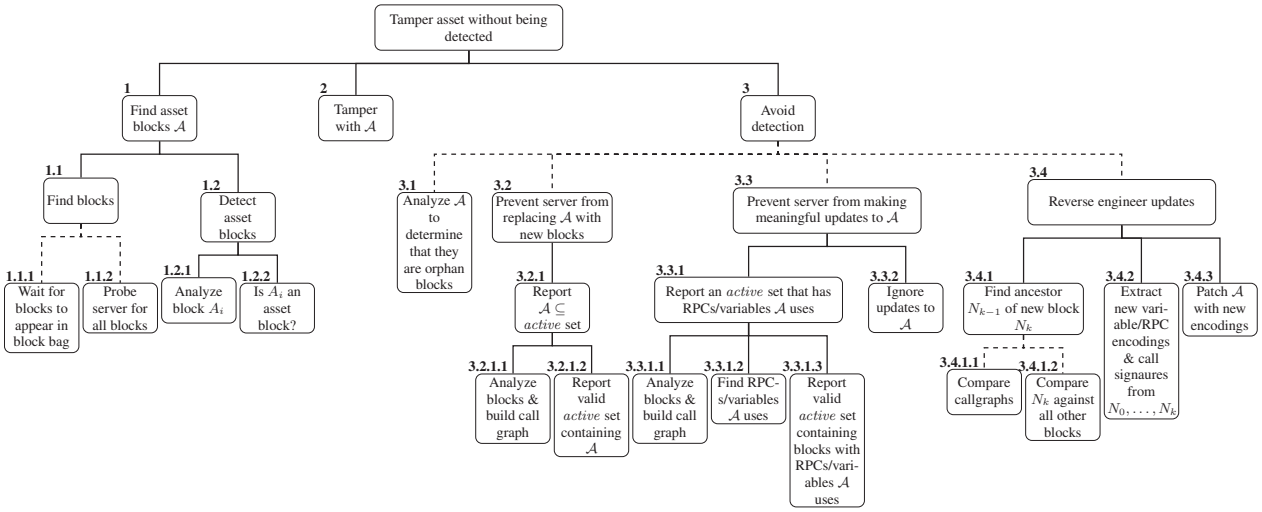


Figure 5: Attack tree [26]. OR-edges are dashed, AND-edges are solid.

by inserting bogus function calls, RPCs, and variable references.

To counter the attacks in node **3.2** and **3.3**, we can again use the **opaque** primitive, inserting calls to non-existing functions. If the adversary reports an *active* set containing such a function, we know that he is cheating.

4.3 Empirical Tests

We implemented three attacks and evaluated our system's ability detect them. In all cases, the same target C program was used, a simple Tetris game with some functionality moved server-side.

An unsophisticated attacker could simply ignore block updates, continuing to execute the tampered program (node **3.3.2** in Figure 5). We simulated this attack by turning off client updates. Since RPCs are frequent in our test program, the server reliably detected the malicious behavior shortly after the first **RPC_encode** update.

A malicious client may attempt to build a snapshot of the entire program, in order to analyze it off-line. This can help with attack nodes **1.1.2, 3.2.1.1,** and **3.3.1.1**. To simulate this attack we engineered a client that disassembles its blocks, looks for blocks which are referenced but not yet held, and requests them from the server. However, this malicious client quickly requested nonexistent blocks to which our system had inserted references; blocks a benign client would never ask for because they were protected by opaque predicates. By adding such bogus calls in just 50% of the blocks in our test program we identified the malicious client after they had successfully requested and received 16 of the 24 real blocks. When we added a call to a bogus function into *every* block the malicious client received only 6 of 24 real blocks (25%) before the attack was detected.

In order to prevent the server from updating blocks a client can report the entire contents of the block bag to the server as his *active* set. This can be seen as a crude example of the attacks in nodes **3.2.1** and **3.3.1** of Figure 5. We implemented such a client and the server was able to use the program call graph to reliably identify the malicious behavior after the *first* update cycle.

4.4 Diversity of Primitive Transformations

The most sophisticated attack in the attack tree is rooted at node **3.4**. The idea is for the attacker to try to identify the history of a new block N_k, i.e. all its ancestors $N_0, \ldots, N_{k-1}$, and to use that information to help him reverse-engineer the transformations that

(a) n-Gram Similarities of Related and Unrelated Blocks

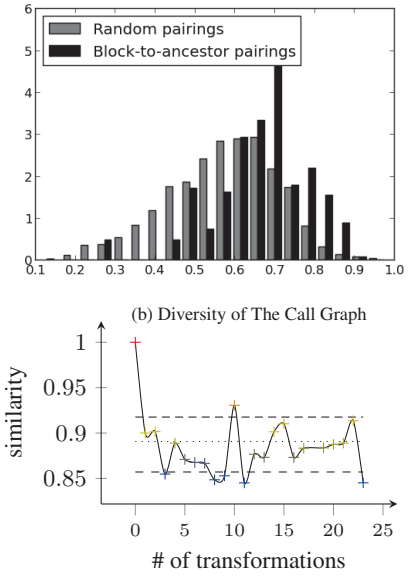


Figure 6: Diversity of blocks and call graphs.

were used to generate N_k. This, in turn, could allow him to intelligently update blocks he has previously tampered with, without having to re-analyze them from scratch.

Determining the ancestor N_{k-1} of a newly updated block N_k can be accomplished in one of two ways: by comparing the call graphs before and after the update (**3.4.1.1**) or by comparing N_k against all blocks in the block bag, looking for a similar block (**3.4.1.2**). We claim that because our primitives generate high entropy in both the call graph and the content of the blocks, recognizing ancestry should be very difficult.

To demonstrate the call graph diversity generated by our system, we repeatedly applied the **merge** and **split** obfuscations to functions from the `gzip` SPEC benchmark. We compared call-graph similarity between the obfuscated and unobfuscated program using

the algorithm from Shang et al. [28]. The data series in Figure 6 (b) shows the similarity scores after each iteration of the transformations. For comparison, the horizontal lines in the graph show similarity scores of `gzip` to other unrelated SPEC programs; this result demonstrates that the call graph of the modified program is often less similar to its ancestor than to that of unrelated code.

Failing to use the call graph to identify N_{k-1}, the attacker may attempt to scan all blocks in his bag and measure their similarity to N_k. We defend against this attack by ensuring that we can generate significantly diverse implementations of a block via obfuscations. To demonstrate this block diversity, we measured n-gram similarity [25] of obfuscated blocks to their ancestors (N_k to N_{k-1} pairings) and compared these scores with the n-gram similarity of unrelated code (N_k to N_j pairings). To simulate the process of a successful decompilation, we first converted block contents with normalized source code, replacing literals and variable names with place-holders and converting the program to high-level intermediate code. The gray bars in Figure 6 (a) show the distribution of n-gram similarities of randomly-chosen unrelated blocks (N_k to N_j pairings). The black bars show the distribution of n-gram similarities of obfuscated blocks to their unobfuscated ancestors (N_k to N_{k-1} pairings). Obfuscations used were interpreter and flatten. The average n-gram similarity of unrelated blocks was .57; the average similarity of a block to its ancestor was .66. Though some obfuscated functions show high similarity to their ancestors, many other block-to-ancestor pairings show low similarity, and would be clearly buried in the noise of all possible block-to-block pairings, making them impractical to find via n-gram analysis.

5. PERFORMANCE EVALUATION

With respect to performance, we evaluate the overhead of our infrastructure (without any primitives applied), the overhead added by individual primitives, and the delay the client experiences as the result of an update. While it would seem interesting to measure the "typical" or average overhead a program might suffer, this is not possible, since it depends as much on the mix of primitives and strategies the scheduler chooses as on the nature of the program itself. For many applications it is more interesting, in fact, to measure *worst case* performance, for example the longest delay a user might suffer as the result of an update. For all measurements we therefore attempt to evaluate the worst case performance overhead for a few of the SPEC benchmarks.

Client measurements were taken with our client running on a laptop with 4 GB of memory and an Intel 2.9 GHz Core i5 processor, running Ubuntu Linux. Server measurements were taken with our server running on an Amazon EC2 `m1.small` instance, providing 1.7 GB of memory and one EC2 compute unit.

Our infrastructure has multiple potential sources of overhead. First of all, our client code `client.c` is a C program that originally might have been compiled as a single large file. Our system instead breaks this into multiple files consisting of a single function, each one compiled separately, and this will impede inter-procedural optimization. Furthermore, an intelligent linker might place related functions of `client.c` on the same physical page, but the client in our system does not see the entire program at once, and cannot make the same choices as easily. A further consequence of continuous updates is that we add a level of indirection to all function calls. To evaluate the overhead that our infrastructure adds, we measured the change in runtime for `bzip`, `gzip`, `mcf` and `crafty`. We found the cost as typically near a five to ten percent runtime increase, though `crafty` was closer to a twenty percent increase.

The latency of an update is the sum of network delay, the time to transform blocks using our primitives, compilation and linking time (since our system is based on CIL which generates C code as output), and dynamic load time on the client. Transformation of blocks is usually on the order of a tenth of a second (even with our most complex primitives). Compilation currently takes roughly a second per function that needed to be changed and transfer across the network takes roughly half of a second per function. The compilation time dominates the cost in our current model, but this could be remedied by compiling multiple files concurrently on separate computation units or by using a system such as LLVM that does not require a final compilation stage.

6. FUTURE WORK

The latency of updates clients suffer in our system is acceptable for some applications (say, medical records databases) but not for those with real-time constraints, such as multiplayer online games. We are investigating multiple ways to reduce the delays, both those caused by network overhead and by code transformations.

Currently, we are working on supporting background generation of working sets. The idea is simple and easily realized using our diversity graph: (1) determine (through profiling or static analysis) active sets that will be commonly occurring on the client and generate block working sets compatible with these active sets in the background; (2) during an update cycle, serve the client with a pre-generated working set if one matches his current active set, otherwise generate one on the fly.

Previous work that has suggested diversification-for-security has typically ignored the correctness issues that follow with randomizing code, or have assumed the existence of extensive test suites that could be run after diversification. Our situation is more difficult than most since we generate code continuously at runtime, under time constraints, meaning comprehensive testing after each transformation is infeasible. However, we do keep track of the history of each generated block through our diversity graph, and we hope to use this information to integrate project unit tests in the system.

7. CONCLUSIONS

We have described a system for detecting tampering of clients running on untrusted nodes in a distributed system. Our system continuously updates client code, keeping it in a state of constant flux, giving the adversary a limited time-window for analyzing and tampering with the code. We employ protocol-preserving code transformations which provide diversity to slow down the adversary's analysis and tampering of the code, and transformations that are not protocol-preserving to make it harder for him to tamper with the code without modifying its expected behavior, thereby making it easier for the trusted server to detect the tampering.

The security afforded by the system is a function of the frequency of code updates and the complexity and variability generated by individual code transformations. This, in turn, is related to the computational overhead a particular program can afford to suffer. To manage overhead, our diversity scheduler is designed to allow detailed control over which parts of the program to transform, which transformations to apply, and uses a rollback technique to avoid compounding too many transformations.

We have shown that our infrastructure itself causes a performance overhead ranging from 4% to 23% and that, for several easy-to-implement attacks, the server reliably and quickly detects the malicious behavior.

8. REFERENCES

[1] B. Anckaert, M. Jakubowski, R. Venkatesan, and K. D. Bosschere. Run-time randomization to mitigate tampering. In *Proceedings of the second International Workshop on Security*, number 4752, pages 153–168, Berlin, Oct. 2007.

[2] B. Anckaert, B. D. Sutter, and K. D. Bosschere. Covert communication through executables. In *Program Acceleration through Application and Architecture driven Code Transformations*, pages 83–85, Sept. 2004.

[3] R. Anderson and M. Kuhn. Low cost attacks on tamper resistant devices. In *IWSP: International Workshop on Security Protocols*, 1997.

[4] D. Aucsmith. Tamper resistant software: An implementation. In R. J. Anderson, editor, *Information Hiding*, pages 317–333. Springer-Verlag, May 1996.

[5] E. G. Barrantes, D. H. Ackley, S. Forrest, and D. Stefanović. Randomized instruction set emulation. *ACM Transactions on Information and System Security (TISSEC)*, 8(1):3–40, Feb. 2005.

[6] M. Ceccato and P. Tonella. Codebender: Remote software protection using orthogonal replacement. *IEEE Software*, 28(2):28–34, 2011.

[7] F. M. Cleveland. Cyber security issues for advanced metering infrasttructure (AMI). In *Power and Energy Society General Meeting - Conversion and Delivery of Electrical Energy in the 21st Century*, pages 1–5. IEEE PES Power Syst. Commun. Comm., July 2008.

[8] F. B. Cohen. Operating system protection through program evolution. `all.net/books/IP/evolve.html`, 1992.

[9] C. Collberg and J. Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley, July 2009.

[10] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *POPL'98*, Jan. 1998.

[11] L. D'Anna, B. Matt, A. Reisse, T. V. Vleck, S. Schwab, and P. LeBlanc. Self-protecting mobile agents obfuscation report — Final report. Technical Report 03-015, NAI Labs, June 2003.

[12] J.-F. Dhem, F. Koeune, P.-A. Leroux, P. Mestré, J.-J. Quisquater, and J.-L. Willems. A practical implementation of the timing attack. In *CARDIS*, pages 167–182, 1998.

[13] M. A. Ertl. Stack caching for interpreters. *SIGPLAN Not.*, 30(6):315–327, 1995.

[14] P. Falcarin, S. D. Carlo, A. Cabutto, N. Garazzino, and D. Barberis. Exploting code mobility for dynamic binary obfuscation. 2011.

[15] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In *HotOS-VI*, pages 67–72, 1997.

[16] A. Haeberlen, P. Aditya, R. Rodrigues, and P. Druschel. Accountable virtual machines, 2010.

[17] G. Hoglund and G. McGraw. *Exploiting Online Games: Cheating Massively Distributed Systems*. Addison-Wesley, 2007.

[18] F. Hohl. Time limited blackbox security: Protecting mobile agents from malicious hosts. In *Mobile Agents and Security*, pages 92–113, 1998.

[19] M. Jakobsson and M. K. Reiter. Discouraging software piracy using software aging. In *DRM Workshop*, pages 1–12, 2001.

[20] G. C. Necula, S. Mcpeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction*, pages 213–228, Grenoble, 2002.

[21] T. Proebsting. Optimizing ANSI C with superoperators. In *POPL'96*. ACM Press, Jan. 1996.

[22] P. R.C. Sandia report: Advanced metering infrastructure security considerations, Nov. 2007.

[23] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a tcg-based integrity measurement architecture. In *SSYM'04*, pages 16–16, 2004.

[24] R. Scandariato, Y. Ofek, P. Falcarin, and M. Baldi. Application-oriented trust in distributed computing. *Availability, Reliability and Security, International Conference on*, 0:434–439, 2008.

[25] S. Schleimer, D. Wilkerson, and A. Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the 2003 SIGMOD Conference*, 2003.

[26] B. Schneier. *Dr. Dobb's Journal*, Dec. 1999.

[27] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. *SIGOPS Oper. Syst. Rev.*, 39(5):1–16, 2005.

[28] S. Shang, N. Zheng, J. Xu, M. Xu, and H. Zhang. Detecting malware variants via function-call graph similarity. In *MALWARE*, pages 113 –120, Oct. 2010.

[29] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. Mutatis mutandis: safe and predictable dynamic software updating. In *POPL '05*, pages 183–194. ACM, 2005.

[30] C. Wang, J. Hill, J. Knight, and J. Davidson. Protection of software-based survivability mechanisms. *Dsn*, 00:0193, 2001.

[31] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. *IEEE Symposium on Reliable Distributed Systems*, 0:260, 2003.

VAMO: Towards a Fully Automated Malware Clustering Validity Analysis

Roberto Perdisci
Dept. of Computer Science
University of Georgia
Athens, GA 30602
perdisci@cs.uga.edu

ManChon U
Dept. of Computer Science
University of Georgia
Athens, GA 30602
manchonu@cs.uga.edu

ABSTRACT

Malware clustering is commonly applied by malware analysts to cope with the increasingly growing number of distinct malware variants collected every day from the Internet. While malware clustering systems can be useful for a variety of applications, assessing the quality of their results is intrinsically hard. In fact, clustering can be viewed as an *unsupervised learning* process over a dataset for which the complete ground truth is usually not available. Previous studies propose to evaluate malware clustering results by leveraging the labels assigned to the malware samples by multiple anti-virus scanners (AVs). However, the methods proposed thus far require a (semi-)manual adjustment and mapping between labels generated by different AVs, and are limited to selecting a *reference sub-set* of samples for which an agreement regarding their labels can be reached across a majority of AVs. This approach may bias the reference set towards "easy to cluster" malware samples, thus potentially resulting in an overoptimistic estimate of the accuracy of the malware clustering results.

In this paper we propose VAMO, a system that provides a *fully automated* quantitative analysis of the validity of malware clustering results. Unlike previous work, VAMO *does not seek a majority voting-based consensus* across different AV labels, and does not discard the malware samples for which such a consensus cannot be reached. Rather, VAMO explicitly deals with the inconsistencies typical of multiple AV labels to build a more representative reference set, compared to majority voting-based approaches. Furthermore, VAMO avoids the need of a (semi-)manual mapping between AV labels from different scanners that was required in previous work. Through an extensive evaluation in a controlled setting and a real-world application, we show that VAMO outperforms majority voting-based approaches, and provides a better way for malware analysts to automatically assess the quality of their malware clustering results.

1. INTRODUCTION

Due to the extensive use of *packing* and other code obfuscation techniques [6], the number of new malware samples collected by anti-virus[1] (AV) vendors has grown enormously in recent years, reaching tens or even hundreds of thousand of new malware samples collected per day (e.g., in 2010 Symantec collected 286 million distinct malware variants [19]). To cope with this increasingly growing number of malware samples and boost the scalability and effectiveness of current malware analysis infrastructures, a number of *malware clustering* and automatic malware categorization systems have been recently proposed [1, 2, 3, 8, 11, 15, 18].

The main objective of malware clustering systems is to group malware samples into *families*, whereby samples that are similar to each other can be considered as variants of the same malware family. Intuitively, malware clustering results can be useful in several ways. For example, new malware samples that are clustered with known malware variants of a given family f may be also categorized as belonging to f. In turn, these newly discovered variants may be used to derive more generic malware detection signatures that have a better chance to match *future variants* of the same family [15]. In addition, malware clustering results may make it easier to identify new, previously unknown malware families [2], or may be used to perform malware triage [11], thus allowing malware experts to select only a small number of variants of a given malware family for manual analysis.

To take full advantage of the above mentioned benefits, malware clustering systems clearly need to be accurate. Unfortunately, it is very challenging to quantitatively assess the accuracy of malware clustering results, because of the lack of reliable ground truth. A common approach to validating the quality of malware clustering results is to compare them to a *reference clustering* obtained by leveraging family labels assigned to the samples by multiple AV scanners [2, 11]. To compensate for inconsistencies in the AV labels, both [2, 18] and [11] use a majority voting approach to select the samples for which an agreement regarding their *AV family* label can be reached. Therefore, a cluster in the reference clustering will include all samples belonging to the same AV family. However, while this approach may appear as a natural choice in absence of complete ground truth, Li et al. [12] have suggested that it may result in an overoptimistic estimate of the malware clustering accuracy. In particular, limiting the reference clustering to samples for which a majority voting-based consensus on the family label can be reached, and discarding the remaining ones, may reduce the reference clusters to only include "easy to cluster" malware samples (i.e., clear-cut cases of malware samples that are very similar to each other) [12], thus potentially causing the accuracy of the malware clustering results to be largely overestimated. In fact, the

[1]While "anti-malware" is probably a more appropriate term, we use "anti-virus" because that is the way in which many vendors of malware scanners and defense solutions still advertise their products.

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

experiments reported in [2] state that among 14,212 malware samples, a majority voting-based consensus could be reached only for 2,658 cases. That is, more than 80% of the samples in the clustering results had to be excluded from the cluster validity analysis.

In this paper we propose VAMO[2], a system that enables an automatic quantitative analysis of the validity of malware clustering results. Like previous work, VAMO leverages the labels assigned to malware samples by multiple AV scanners to construct a reference clustering. However, unlike previous work, VAMO *does not seek a majority voting-based consensus*, and does not discard the samples for which such a consensus cannot be reached. Rather, VAMO explicitly deals with (and aims to mitigate the effect of) the inconsistencies typical of the AV labels to build a more representative reference clustering. Furthermore, VAMO avoids the need of a (semi) manual mapping between AV labels from different scanners that was required in previous work (notice that while some efforts exist to standardize the "language" used to assign the AV labels (e.g., `http://maec.mitre.org`), so far they have not been successful). Also, we would like to emphasize that while AV labels suffer from some limitations, as we discuss more in detail in Section 7, they are used as a reference by many researchers because it is hard to obtain a more accurate ground truth for datasets containing tens of thousands of malware samples.

VAMO leverages historic malware archives and the related multiple AV labels to *learn an AV Label Graph* (see Figure 1). An *AV Label Graph* (see Section 5.1) is defined as an undirected weighted graph, which aims to: (1) automatically learn the mapping between malware family names assigned by different AVs, thus avoiding the need to manually build or adjust such mappings; (2) identify cases in which one (or more) AV scanners tend to inconsistently use several family names to label samples that belong to the same family according to other competitors' scanners; (3) learn the *level of similarity* between AV labels assigned by different AV scanners, by looking at the number of times that certain malware family labels are jointly assigned to the same samples. While the concept of *AV Label Graph* was first introduced in [15], here we refine its definition and use it in the context of our novel VAMO system. Also, it is worth noting that the *AV Label Graph* is only one component of the entire VAMO system.

Learning the *AV Label Graph* enables us to measure the similarity between malware samples in a dataset based purely on their AV labels (see Section 5.1 for details). As shown in Figure 1, given a malware dataset $\mathbf{M}$ and the related multiple AV labels assigned to its malware samples, we can (a) apply a third-party malware clustering algorithm (e.g., [2, 11, 15, 18]) on $\mathbf{M}$ to partition it in a number of malware clusters, (b) use VAMO to build a reference clustering for $\mathbf{M}$ using similarities among its samples measured according to their AV labels, and (c) compute the *level of agreement* between VAMO's reference clustering and the third-party malware clustering results, thus quantitatively assessing their quality.

In summary, this paper makes the following contributions:

- We propose a novel system, called VAMO, that enables a *fully automated* malware clustering validity analysis.
- We perform an extensive evaluation of how different types of AV label inconsistencies may negatively impact a validity analysis performed via majority voting-based approaches, and show the advantages that VAMO brings over previous work.
- We perform experiments with real-world malware archives, and demonstrate how VAMO can be applied in practice to assess the quality of malware clustering results over large malware datasets.

[2]Validity Analysis of Malware-clustering Outputs.

2. RELATED WORK

Cluster Validity Analysis Besides the clustering validity indexes reported in Halkidi et al.'s survey [7], which we summarize in Section 3.2, a number of alternative validity indexes have been proposed. In [17], Rendon et al. present a comparison of internal and external clustering validity indexes, while Meilă [14] and Pfitzner et al. [16] introduce a number of new metrics to compare two different clusterings. In [5], Fowlkes and Mallows introduce a measure of similarity between two hierarchical clusterings obtained by cutting the two dendrograms at heights h_1 and h_2, respectively, which yield the same number of clusters k. Than, for each value of k, the number of matching entries from the two different clusterings are counted to obtain a measure of comparison. Our approach to cluster validity analysis (Section 5) is inspired by [5]. However, our method does not focus on comparing different hierarchical clusterings. Rather, VAMO leverages hierarchical clustering to generate a *reference clustering dendrogram*, and compares third-party clustering results to this dendrogram by finding the cut height h that yields the maximum agreement between the third-party results and VAMO's reference clustering.

Malware Clustering Bailey et al. [1] presented one of the first studies on behavior-based malware clustering. Furthermore, in [1] the authors presented a quantitative analysis of the inconsistency in the labels assigned by different AVs. Bayer et al. [2] introduced a much more scalable way to perform behavior-based malware clustering. In addition, they proposed to validate their clustering results by comparing them against a clustering obtained using a majority voting-based approach over multiple malware family labels assigned to the samples by six different AVs [2]. A similar validation approach was used in [18]. In [8], Hu et al. perform malware clustering using static analysis, instead of behavior-based features, by leveraging function-call graphs, while [11] introduces a system called BitShred that aims to improve scalability in malware clustering systems.

In [12], Li et al. discuss a number of challenges related to the evaluation of results generated by malware clustering systems. In particular, by using plagiarism detection algorithms to measure the similarity between malware samples, they show that a factor contributing to the strong results reported in [2] might be that the 2,658 validation instances selected via majority voting on multiple AVs are simply easy to classify. However, no complete solution is offered on how to perform a better malware clustering validity analysis. Our work is a step forward towards such a solution.

While most malware clustering systems are based on system-level behavior or static-analysis-based features, [15] proposed a malware clustering system that focuses on the network behavior of malware and introduced the concept of AV Label Graph, which we refine and use in this paper in the context of VAMO. It is worth noting that the use of AV Label Graphs in [15] is significantly different from this paper. Previous work did not present a comprehensive malware clustering validity analysis system, and the cohesion and separation validity indexes used in [15] were mainly *internal* validity indexes that required a significant amount of interpretation through manual analysis. On the other hand, VAMO introduces a comprehensive, fully automated malware clustering validity analysis process that can more readily be used to select the parameters of a malware clustering system, or to compare results obtained using different clustering algorithms.

	AV1	AV2	AV3	AV4
Detected samples	590,341	825,766	702,124	1,030,354
Detection rate (%)	53.3%	74.5%	63.4%	93.0%
Distinct AV labels	20,217	15,138	2,208	175,333
Distinct family labels	3,330	4,729	1,710	3,520
Distinct first variants	20,217	13,851	2,199	51,732

Table 1: AV labels for a dataset of 1,108,289 distinct malware samples.

3. BACKGROUND

In this Section, we first provide quantitative information regarding the inconsistency typical of multiple AV labels. Then, we discuss the background concepts that we will use to perform automated clustering validity analysis.

3.1 Measuring Inconsistency in AV Labels

In this Section, we aim to quantify the "inconsistency" typical of multiple AV labels that has been *qualitatively* discussed in previous work [2, 15, 18], and analyzed more in details in [1, 13]. Our main goal is to suggest that (semi-) manually creating a mapping between malware family labels and correct the inconsistent (or erroneous) labels, which was required in previous work to perform malware cluster validity analysis (e.g., in [2]), is in fact a fairly difficult task. In addition, we show that in a large number of cases no majority voting-based consensus can be reached. Our results confirm previous findings [1] by using a more recent and much larger malware dataset.

To this end, we performed a number of measurements over a large dataset of AV labels assigned by four different major AV vendors (namely, Symantec, McAffee, Avira, and Trend Micro) to a set of 1,108,289 distinct malware samples[3]. These malware samples were collected from different sources over the course of one entire year, from 2011-01-01 to 2011-12-31 (it is worth noting that we only consider malware samples that were detected as such by at least one out of the four AV scanners). The AVs used to scan the samples were updated daily, and each malware sample was scanned with each AV once a day for 30 days[4], starting from the day in which the sample was collected. In the following, we will refer to the four AV scanners, in no particular order, as `AV1`, `AV2`, `AV3`, and `AV4`. We intentionally mask the specific AV vendor names, when reporting the results, to avoid controversy (the results we report may be seen as damaging to one or more vendors, due to their low detection rate). After all, we do not intend to establish what vendor performs the best over our malware dataset. Rather, we focus on the inconsistencies in the malware labels, both within a given AV vendor as well as across vendors.

3.1.1 Overview

Table 1 summarizes our AV label dataset. As we can see, the detection rate, number of distinct (complete) labels, and the number of distinct malware family labels varies greatly across the different AV scanners. For example, `AV3` assigned a label to 702,124 (63.4%) malware samples, but the number of distinct labels was only 2,208. This means that, in average, the same label was assigned to 317 different samples. This behavior is very different from the other AVs, and in particular from `AV4` for which in average the same label was assigned to (approximately) six samples. In addition, among the 1,108,289 distinct malware samples, only 420,920 (38%) were labeled (i.e., detected) by more than two different AVs. This suggests that because a majority voting approach would require three out of four AVs to agree on the labels (two out of four would only represent a tie), in our example scenario no majority voting-based consensus can be reached on the correct malware family label for at least 38% of the samples. This problem is exacerbated by the fact that even in the cases in which three or more labels are available, the AVs may not agree on the family those samples belong to, as we discuss in Section 3.1.2

3.1.2 Family Labels

We now focus on malware family names, rather than considering full AV labels. We will consider the malware cluster Clust. 1 shown below as an example, to explain how we derive the malware family names. This malware cluster was obtained using [15]. In Clust. 1, each row represents a malware sample (indexed by the last four bytes of its `MD5` sum), and reports the labels assigned to the sample by three different AVs, namely McAfee (`M`), Avira (`A`), and Trend Micro (`T`).

Clust. 1 Malware cluster with inconsistent AV labels.

b1b6da81	M=W32/Virut.gen	A=TR/Drop.VB.DU.1	T=PE_VIRUT.XO-1
ec34ca31	M=W32/Virut.gen	A=TR/Drop.VB.DU.1	T=PE_VIRUT.XO-1
c2276216	M=W32/Virut.gen	A=W32/Virut.E.dam	T=PE_VIRUT.NS-4
089ae4f5	M=W32/Virut.gen	A=W32/Virut.AX	T=PE_VIRUT.D-1
8ba552c9	M=W32/Virut.gen	A=TR/Drop.VB.DU.1	T=PE_VIRUT.XO-1
8cb0ab6c	M=W32/Virut.gen	A=WORM/Korgo.U	T=PE_VIRUT.D-4
b0b75f70	M=W32/Virut.gen	A=W32/Virut.X	T=PE_VIRUT.XO-1
a306b4e7	M=W32/Virut.gen	A=W32/Virut.Gen	T=PE_VIRUT.D-1
337a2cf4	M=W32/Virut.gen	A=W32/Virut.Gen	T=PE_VIRUT.D-1
62d18c7e	M=W32/Virut.gen	A=W32/Virut.Gen	T=PE_VIRUT.D-1
8dbca633	M=W32/Virut.gen	A=TR/Drop.VB.DU.1	T=PE_VIRUT.XO-1
ac433383	M=W32/Virut.n	A=W32/Virut.Gen	T=PE_VIRUX.A-3
cae61d9e	M=W32/Virut.gen	A=W32/Virut.X	T=PE_VIRUT.XO-2
7cc795f1	M=W32/Virut.gen	A=W32/Virut.Gen	T=PE_VIRUT.D-1
8de5214b	M=W32/Virut.gen.a	A=W32/Virut.AM	T=PE_VIRUT.XY
4d26cb0a	M=W32/Virut.gen	A=W32/Virut.Gen	T=PE_VIRUT.D-1
9fb75631	M=W32/Virut.n	A=W32/Virut.Gen	T=PE_VIRUX.A-3
229004b9	M=W32/Virut.gen	A=W32/Virut.X	T=PE_VIRUT.XO-1
28a85d8a	M=W32/Virut.gen	A=TR/Drop.VB.DU.1	T=PE_VIRUT.XO-1
663c5f6c	M=W32/Virut.d	A=W32/Virut.Z	T=PE_VIRUT.GEN-2
de6f1e00	M=	A=W32/Virut.Gen	T=PE_VIRUT.D-4
1ff43bca	M=	A=W32/Virut.X	T=PE_VIRUT.XO-4
ea580f6d	M=W32/Virut.n	A=W32/Virut.Gen	T=PE_VIRUX.A-3
a844eeff	M=W32/Virut.gen	A=TR/Drop.VB.DU.1	T=PE_VIRUT.XO-1
4f8613fd	M=W32/Virut.gen	A=TR/Drop.VB.DU.1	T=PE_VIRUT.XO-1

To derive the malware family name, we split each label into substrings divided by the '.' symbol, and we extract the first substring. For example, `W32/Virut.gen` becomes `W32/Virut` (Symantec uses a slightly different notation, compared to the other AV vendors. To extract the family label from Symantec's labels, we consider the first two substrings obtained by splitting the labels by the '.' symbol. For example, `W32.Sality.AE` would become `W32.Sality`).

As we can see from Clust. 1, in this case both McAfee and Trend Micro are very consistent, because they label the vast majority of the samples as belonging to the *Virut* malware family, with the exception of two samples that were missed by McAfee and three samples that are labeled as `PE_VIRUX` (rather than `PE_VIRUT`) by Trend Micro. On the other hand, Avira is much less consistent, because it assigned three different family names to the samples (i.e., `TR/Drop`, `W32/Virut`, `WORM/Korgo`).

Table 1 reports the total number, per AV, of distinct family names obtained from all labels in our datasets. Also, Table 1 reports the total number, per AV, of distinct "first variant" labels, i.e., labels obtained by combining the first two label substrings (the first three, in case of Symantec). Again, there is a relatively large difference between the numbers obtained from different AVs.

To measure the number of common family names per sample

[3]This dataset was kindly provided by a well-known security company.

[4]If an AV scanner `AVi` detected a sample m and assigned it a label on day $d < 30$, the data collector would stop scanning m with `AVi` for the remaining days, but continued scanning the sample with the other AVs until they also assigned a label or $d > 30$.

across different AVs, we further normalized the family names, for example by cutting the label prefix (e.g., `W32/`, `PE_`, etc.) and reducing all labels to lower case. For example, the first sample in Clust. 1 would be labeled as {`virut`, `drop`, `virut`}. This was done to maximize the number of common family names we could find for a given sample across different AVs. Even after this normalization, we could find a common family label across at least three out of four AVs for only 2.4% of the samples, and a common label across at least two out of four AVs for only 5.6% of the samples. Performing a manual mapping between the labels to mitigate the effect of different "terminology" used by different AVs may improve on these results. However, even after such manual mapping a majority voting-based consensus between the AVs cannot be reached for the vast majority of the samples. This findings are consistent with the experiments conducted in [2], in which a majority voting-based consensus could be reached only for less than 20% of the samples. Therefore, a reference clustering generated via majority voting may miss to represent a large portion of the malware dataset, causing a potential overestimate of the clustering quality, as also suggested in [12].

3.2 Validity Indexes

Clustering can be viewed as an *unsupervised learning* process over a dataset for which the complete ground truth is usually not available. Therefore, unlike in supervised learning settings, analyzing the *validity* of the clustering results is intrinsically hard. The assessment of the quality of clustering results often involves the use of subjective criteria of optimality [10], which are typically application specific, and commonly involves extensive manual analysis by domain experts. To aid the clustering validation process, a number of methods and quality indexes have been proposed [7, 9]. Halkidi et al. [7] provide a survey of cluster validity analysis techniques, which aim to evaluate the clustering results to *find the partitioning that best fits the underlying data*.

Three main *cluster validity* approaches are described [7]: (1) *external criteria* evaluate the clustering results by comparing them to a pre-specified structure, or *reference clustering*; (2) *internal criteria* rely solely on quantities derived from the data vectors in the clustered dataset (e.g., using a proximity matrix, and computing quantities such as inter- and intra-cluster distances); (3) *relative criteria* compare clustering results obtained using the same clustering algorithm with different parameter settings, to identify the best parameter configuration.

External validation criteria are particular attractive, because they offer a quantitative way to measure the *level of agreement* between the obtained clustering results and a reference clustering that is considered to be the ground truth [7, 16]. However, the main problem is exactly how to construct the reference clustering in the first place. This is one of the problems we address in this paper: building a reference clustering that can be used for validating the results of malware clustering systems.

Assuming a reference clustering is available, different external validity indexes can be used for measuring the quality of the clustering results. We briefly describe some of them below. Let $\mathcal{M}$ be our dataset, $\mathbf{Rc} = \{Rc_1, .., Rc_s\}$ be the set of *s reference clusters*, and $\mathbf{C} = \{C_1, .., C_n\}$ be our clustering results over $\mathcal{M}$. Given a pair of data samples (m_1, m_2), with $m_1, m_2 \in \mathcal{M}$, we can compute the following quantities:

- a is the number of pairs (m_1, m_2) for which if both samples belong to the same reference cluster Rc_i, they also belong to the same cluster C_j.
- b is the number of pairs (m_1, m_2) for which both samples belong to the same reference cluster Rc_i, but are assigned to two different clusters C_k and C_h.
- c is the number of pairs (m_1, m_2) for which both samples belong to the same cluster C_i, but are assigned to two different reference clusters Rc_k and Rc_h.
- d is the number of pairs (m_1, m_2) for which if the samples belong to two different reference clusters Rc_i and Rc_j, they also belong to different clusters C_l and C_m.

Based on the above definitions, we can compute the following external cluster validity indexes [7]:

- **Rand Statistic**. $RS = \frac{a+d}{a+b+c+d} = \frac{a+d}{|\mathcal{M}|}$
- **Jaccard Coefficient**. $JC = \frac{a}{a+b+c}$
- **Folkes and Mallows Index**. $FM = \frac{a}{\sqrt{(a+b)(a+c)}}$

For all three indexes above, which take values in $[0, 1]$, higher values indicate a closer similarity between the clustering $\mathbf{C}$ and the reference clustering $\mathbf{Rc}$.

The authors of [2, 18], proposed to use different indexes, based on *precision* and *recall*, to measure the level of agreement between behavior-based malware clustering results $\mathbf{C}$ and a (semi-)manually generated reference clustering $\mathbf{Rc}$ derived by using majority voting over multiple AV labels. In this setting, precision and recall, and the related $F1$ index, are defined as follows:

- **Precision**. $Prec = 1/n \cdot \sum_{j=1}^{n} \max_{k=1,..,s}(|C_j \cap Rc_k|)$
- **Recall**. $Rec = 1/s \cdot \sum_{k=1}^{s} \max_{j=1,..,n}(|C_j \cap Rc_k|)$
- **F1 Index**. $F1 = 2\frac{Prec \cdot Rrec}{Prec + Rrec}$

In the remainder of the paper, we will often refer to the external validity indexes defined above.

4. SYSTEM OVERVIEW

Figure 1 provides a high-level overview of VAMO. We assume that a third-party has employed a malware clustering system, for example one of the systems proposed in [2, 11, 15], to partition a malware dataset $\mathbf{M}$ into a set of clusters $\mathbf{C} = \{C_1, C_2, .., C_x\}$, with $\bigcup_{i=1}^{x} C_i = \mathbf{M}$. VAMO's objective is to validate the quality of $\mathbf{C}$ (i.e., the malware clustering results). We now provide a description of VAMO's components shown in Figure 1.

AV Label Dataset Given a large *historic archive dataset* of malware samples $\mathcal{A}$ (which is different from $\mathbf{M}$), we first collect the set of family labels assigned by multiple AV scanners to each of the malware samples $m_k \in \mathcal{A}$. The resulting AV labels dataset can be represented as a set of tuples $\mathcal{L} = \{(l_{k,1}, l_{k,2}, .., l_{k,\nu})\}_{k=1..n}$, where $l_{k,i}$ is the malware family label assigned by the i-th of ν AV scanners to malware sample m_k, with $k = 1, .., n$, and $n = |\mathcal{A}|$. If an AV scanner misses to detect a malware sample, the related label in the set $\mathcal{L}$ will be assigned a *unique* placeholder "unknown" family label. It is worth noting that the malware dataset $\mathcal{A}$ need not contain actual executable malware samples. In fact, $\mathcal{A}$ may simply contain a list of hashes (e.g., `md5` or `sha1`) computed by a third party (e.g., the owner of a large malware dataset who cannot share the malware itself) over known malware samples. In this case, the label dataset $\mathcal{L}$ may be obtained by querying a service such as `virustotal.com` to obtain, for each hash, the related malware family labels from multiple AV scanners.

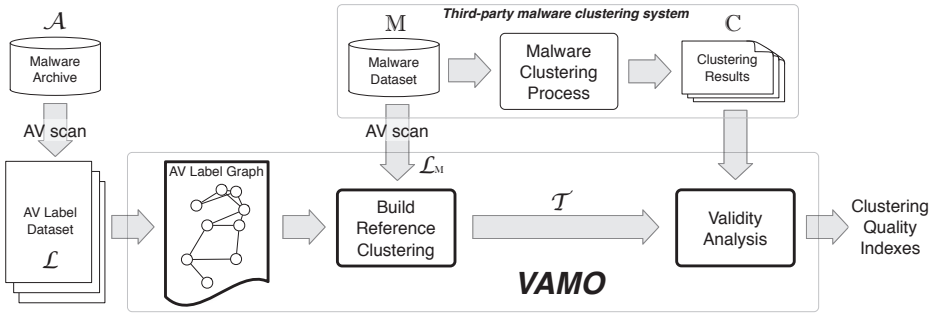


Figure 1: VAMO System Overview

AV Label Graph VAMO uses the label dataset $\mathcal{L}$ to *learn* an *AV Label Graph* (defined formally in Section 5.). Basically, a node in the graph represents a malware family name attributed by a certain AV scanner (or AV, for short) to one or more malware samples in $\mathcal{A}$. For example, assuming the i-th AV assigned the label `family_x` to at least one malware sample, the *AV Label Graph* will contain a node called `AVi_family_x`. Two nodes, say `AVi_family_x` and `AVj_family_y`, will be connected by an undirected edge if there exists at least one malware sample $m_k \in \mathcal{A}$ that has been assigned label `family_x` by the i-th AV, and `family_y` by the j-th AV, respectively. Each edge is assigned a weight that depends on the number of times that the connected nodes (i.e., the connected labels) were assigned to a same malware sample. Notice that if the i-th AV missed to detect a given malware sample, the related missing label will be replace by a label such as `AVi_unknown_U`, where `U` is a unique identifier.

Reference Clustering Similarly to what we did with $\mathcal{A}$, given the malware dataset $\mathbf{M}$ (i.e., the input to the third-party clustering system), we first collect the set of labels assigned by ν different AVs to each of the malware samples $m_k \in \mathbf{M}$, thus obtaining a dataset $\mathcal{L}_M$ consisting of a tuple (or vector) of family labels $L_k = (l_{k,1}, l_{k,2}, .., l_{k,\nu})$ per each sample m_k. At this point, we leverage the previously learned *AV Label Graph* to measure the *dissimilarity* (or distance) between samples in $\mathbf{M}$ according to their malware family labels. Specifically, we measure the distance between two malware samples $m_i, m_j \in \mathbf{M}$ by measuring the distance between their respective label vectors L_i and L_j in the graph. We give a formal definition of label-based distance between malware samples in Section 5.1. At a high level, we compute the distance between two samples m_i, m_j by computing the median among the shortest paths in the *AV Label Graph* between all paris of labels l_k, l_h, with $l_k \in L_i$ and $l_h \in L_j$. This allows us to compute an $r \times r$ distance matrix $\mathbf{D}$, where $r = |\mathbf{M}|$ and element $\mathbf{D}[i, j]$ is the distance between samples m_i, m_j. The final reference clustering is obtained by applying average-linkage hierarchical clustering [9, 10] on the distance matrix $\mathbf{D}$. The result is not an actual partitioning of the malware dataset $\mathbf{M}$. Rather, the reference clustering is represented by a *dendrogram* [9], i.e., a tree-like data structure that expresses the "relationship" between malware samples. Cutting this dendrogram at any particular height would produce a partitioning of $\mathbf{M}$ according to the AV label-based distances (see Section 5 for details).

Validity Analysis Let $\mathcal{T}$ be the reference clustering dendrogram output by the previous step. The *Validity Analysis* module takes in input $\mathcal{T}$ and the set of malware clusters $\mathbf{C}$ output by the third-party malware clustering system. At this point, VAMO applies the *external* validity indexes introduced in Section 3 to compute the maximum level of agreement between $\mathbf{C}$ and all possible reference clusterings obtained by cutting $\mathcal{T}$ at different heights. For example, we can compute the maximum Jaccard coefficient $\hat{J}$ between all possible reference clusterings and $\mathbf{C}$. The higher $\hat{J}$, the stronger the agreement between $\mathbf{C}$ and the AV label-based reference clustering.

Effectively, VAMO compares the third-party clustering results $\mathbf{C}$ to a reference clustering obtained by partitioning the dataset $\mathbf{M}$ according to the relationships among multiple AV labels learned from the archive malware dataset $\mathcal{A}$. It is worth noting that this process has some similarities with the majority voting-based approach used in previous work. In fact, the effect of the majority voting approach is to group a subset of the malware in $\mathbf{M}$ according to the labels assigned by multiple AVs to the samples in the very same $\mathbf{M}$ dataset. VAMO is different because (a) it automatically learns the relationships among malware family labels assigned by different AVs, and does not require any manual (or semi-manual) mapping between them; (b) it introduces a measure of label-based distance between malware samples that is not limited to the cases in which a majority voting-based consensus can be achieved; (c) it enables the computation of well known external validity indexes over the entirety of malware clustering results, rather than focusing only on "easy-to-cluster" subset of the malware dataset. In Section 6.1 we empirically show that building a reference clustering based on the *AV Label Graph* and applying the validity analysis process outlined above outperforms the majority voting-based cluster validation approach proposed in previous work.

5. VALIDITY ANALYSIS

In this Section, we provide more details on how VAMO builds the reference clustering by leveraging multiple AV labels, and how the clustering validity indexes are computed to compare third-party malware clustering results to VAMO's reference clustering.

5.1 Building a Reference Clustering

As mentioned in Section 4, the first step to obtaining the reference clustering is to build an *AV Labels Graph*. This graph expresses the "relationships" between different AV labels, and *automatically learns* the likelihood that different labels from different AVs will be assigned to the same malware sample, based on historic observations.

Assume $\mathbf{M}$ is the malware dataset used as input to a third-party (e.g., behavior-based) malware clustering system, as shown in Fig-

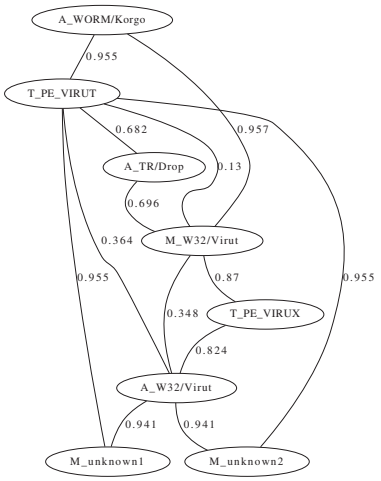


Figure 2: AV Label Graph for Clust.1 (see Section 3.1)

ure 1. Also, let $\mathcal{A}$ be a large historic malware archive containing, for example, malware samples collected during the past several months, and that $\mathbf{M} \subset \mathcal{A}$ (i.e., $\mathcal{A}$ contains all the "current" samples collected in $\mathbf{M}$, plus a large set of malware samples collected in the past). We define an *AV Label Graph* learned from $\mathcal{A}$ as follows.

DEFINITION 1. - ***AV Label Graph***. *An AV Label Graph is an undirected weighted graph. Given an archive of n malware samples $\mathcal{A} = \{m_i\}_{i=1..n}$, let $\mathcal{L} = \{L_1 = (l_1, .., l_\nu)_1, .., L_n = (l_1, .., l_\nu)_n\}$ be a set of label vectors, where a label vector $L_h = (l_1, .., l_\nu)_h$ is an ordered set of malware family labels assigned by ν different AV scanners to malware $m_h \in \mathcal{A}$. The AV Label Graph $\mathcal{G} = \{V_k, E_{k_1,k_2}\}_{k=1..l}$ is constructed by adding a node V_k for each distinct label $l_k \in \mathcal{L}$. Two nodes V_{k_1} and V_{k_2} are connected by a weighted edge E_{k_1,k_2} if the labels l_{k_1} and l_{k_2} related to the two nodes appear at least once in the same label vector $L_h \in \mathcal{L}$ (that is, if they are both assigned to a malware sample m_h). Each edge E_{k_1,k_2} is assigned a weight $w = 1 - \frac{m}{\max(n_1,n_2)}$, where n_1 is the number of label vectors $L_h \in \mathcal{L}$ that contain l_{k_1}, n_2 is the number of vectors that contain l_{k_2}, and m is equal to the number of vectors containing both l_{k_1} and l_{k_2}.*

For example, assume $\mathcal{A}$ contains all (and only) the samples shown in Clust. 1 (shown in Section 3). In this case, the related *AV Label Graph* is shown in Figure 2. Notice that in reality $\mathcal{A}$ will typically contain thousands of samples, and that the graph in Figure 2 is reported simply to provide an example of how the *AV Label Graph* is computed. Also, notice that the missing labels were replaced with *unique* "unknown" identifiers.

Once the *AV Label Graph* is computed, we build a reference clustering *dendrogram* as follows (notice that a dendrogram is a tree-like data structure generated by hierarchical clustering [9]). Given any two samples $m_i, m_j \in \mathbf{M}$ we first "map" each sample onto the graph, and then compute the distance $d_{i,j}$ between m_i, m_j on the graph, thus obtaining a distance matrix $\mathbf{D}$ in which $\mathbf{D}[i,j] = d_{i,j}$. A more formal definition of graph-based distance between malware samples is given below.

DEFINITION 2. - ***Graph-based Distance***. *Let $m_i \in \mathbf{M}$ be a malware sample, and $L_i = (l_1, .., l_\nu)_i$ be its label vector. By definition, each label $l_{h,i} \in L_i$ corresponds to a node $V_{h,i}$ in the AV Label Graph, with $h = 1, .., \nu$. Therefore, sample m_i can be mapped to a list $\mathbf{V}_i = (V_{1,i}, .., V_{\nu,i})$ of ν nodes in the graph. Now, let $\mathbf{V}_i$ and $\mathbf{V}_j$ be the lists of nodes related to m_i and m_j, respectively. To compute the distance $d_{i,j}$ between m_i, m_j, we first compute the length of the shortest path p_k among a pair of nodes $(V_{k,i}, V_{k,j})$, for each $k = 1, .., \nu$. Then, we compute $d_{i,j}$ as the median among all p_k, with $k = 1, .., \nu$.*

After computing the distance matrix $\mathbf{D}$, we apply average-linkage hierarchical clustering, which outputs a *dedrogram* $\mathcal{T}$ that expresses the "relationship" between the malware samples in $\mathbf{M}$ according to their AV labels. Section 5.2 explains in details how the reference clustering dendrogram $\mathcal{T}$ can be used to validate third-party clustering results.

5.2 Computing the Validity Indexes

As mentioned above, by cutting the reference clustering dendrogram $\mathcal{T}$ at a given height h, we obtain an actual partitioning of the dataset $\mathbf{M}$ into a set of reference clusters $\mathbf{Rc} = \{Rc_1, .., Rc_w\}$. Then, the *level of agreement* between $\mathbf{Rc}$ and the third-party clustering results $\mathbf{C} = \{C_1, C_2, .., C_x\}$ can be computed using the external validity indexes introduced in Section 3.2. Naturally, different values of h will produce a different set of reference clusters, and therefore the values of these validity indexes will also differ. Therefore, to decide where exactly to cut the dendrogram $\mathcal{T}$ we proceed as follows. Let $\mathbf{Rc}(h)$ be the set of reference clusters obtained by cutting $\mathcal{T}$ at height h. Also, assume $I(\mathbf{Rc}(h), \mathbf{C})$ is an external validity index computed over the clusterings $\mathbf{Rc}(h)$ and $\mathbf{C}$ (e.g., $I(\cdot)$ could be equal to the Jaccard index, or one of the other indexes outlined in Section 3.2). We then cut $\mathcal{T}$ at height $h^* = \text{argmax}_h\{I(\mathbf{Rc}(h), \mathbf{C})\}$, so that h^* is the cut at which the level of agreement between $\mathbf{C}$ and the VAMO's reference clustering is maximum.

In summary, we perform hierarchical clustering of the malware samples in $\mathbf{M}$ according to similarities in their AV labels by leveraging the previously learned *AV Label Graph*, and then we find the set of reference clusters $\mathbf{Rc}(h^*)$ that *best explains* (or *agrees* with) the third-party clustering results $\mathbf{C}$. This is useful because given two different third-party results $\mathbf{C_1}$ and $\mathbf{C_2}$ (e.g., given by the same behavior-based malware clustering systems configured with different parameter values, or given by different malware clustering systems), VAMO allows us to establish which of them has the *highest level of agreement with the underlying multiple AV labels*.

6. EVALUATION

6.1 VAMO v.s. Majority Voting

In this Section, we present a set of experiments performed in a controlled setting. Our objective is to show that, when faced with noisy AV labels, VAMO outperforms majority voting-based approaches. Namely, in the vast majority of cases VAMO produces an AV label-based reference clustering that better explains (or agrees with) the *true* malware clusters. To this end, we use the following high-level approach. We simulate a controlled dataset of malware samples for which we know exactly what samples should belong to what malware cluster, and first assume that all samples are perfectly (i.e., correctly) labeled by multiple AVs. Then, we gradually introduce more and more noise into the AV labels, thus simulating the inconsistent labeling typical of real-world AVs (see Section 3.1). For each noise increase, we apply both VAMO and a majority voting-based approach to obtain an AV label-based reference clustering, and the obtained results show that VAMO's reference clustering yields validity indexes that offer a higher level of

agreement with the *true* malware clusters, compared to using majority voting.

6.1.1 Controlled Datasets

We create a synthetic dataset to simulate a scenario in which we have a historic archive $\mathcal{A}$ consisting of 3,000 distinct malware samples and the related dataset $\mathcal{L}$ of labels assigned by three different AV scanners to each of these 3,000 samples. Furthermore, we create a dataset $\mathbf{M}$ containing 300 distinct samples, with $\mathbf{M} \subset \mathcal{A}$ (i.e., $\mathbf{M}$ is a proper subset of $\mathcal{A}$). Therefore, the label dataset $\mathcal{L}_M$ containing the AV labels for the malware samples in $\mathbf{M}$ can be directly obtained from $\mathcal{L}$ (since $\mathbf{M} \subset \mathcal{A}$, then $\mathcal{L}_M \subset \mathcal{L}$). It is worth noting that we named these datasets following the same terminology that we used in Section 4 and in Figure 1.

At first, we assume to have perfect knowledge (i.e., perfect *ground truth*) regarding the malware family each sample belongs to. Specifically, we construct the datasets so that the samples in $\mathcal{A}$ (and the related malware labels in $\mathcal{L}$) belong to 15 different malware families, with 200 samples per family, and that each of the three AVs consistently assigns the correct malware family name to the samples in $\mathcal{A}$, and therefore also to the samples in $\mathbf{M}$. In practice, to obtain $\mathbf{M}$ we simply randomly (uniformly) select 300 samples from $\mathcal{A}$. Also, since we know exactly what malware belong to what family, we can precisely partition the dataset $\mathbf{M}$ into a set of 15 malware clusters $\mathbf{C} = \{C_1, C_2, ..., C_s\}$, with $s = 15$.

It is worth noting that in this idealized scenario we also assume the AVs use the very same family names for the malware family labels. In other words, we assume the AVs all agree on using the same terminology or notation. This means that no manual mapping between family names assigned by different AVs is needed, and a majority voting-based approach can be applied directly. This typically does not hold in practice, in which case we would need to obtain the name mapping before being able to apply majority voting. On the other hand, VAMO is agnostic to differences in the terminology that the AVs use to assign malware family names, because VAMO will automatically learn the relationships between different malware family names through the *AV Label Graph* construction, as discussed in Section 4 and Section 5.

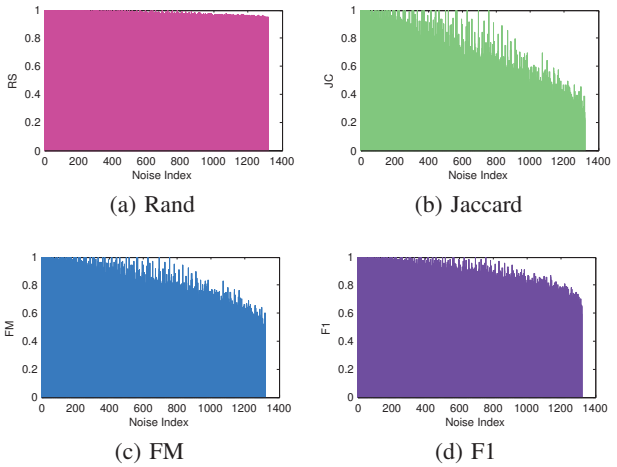
6.1.2 Simulating Inconsistency in the AV Labels

To simulate inconsistency in the AV labels, we proceed as follows. We start from the label dataset $\mathcal{L}$ described above, and we progressively inject more and more noise into the labels. Specifically we inject the following two types of noise:

- **Label Flips** Given a malware $m_k \in \mathcal{A}$, and its label vector $L_k = (l_{1,k}, l_{2,k}, l_{3,k}) \in \mathcal{L}$, with probability p'_f we replace label $l_{\nu,k}$ with a different label $l'_{\nu,k}$ chosen among the 14 other possible malware family labels, where the probability p'_f is a preset "probability of flip".

- **Missing Labels** Similarly, given a malware $m_k \in \mathcal{A}$, and its label vector $L_k = (l_{1,k}, l_{2,k}, l_{3,k}) \in \mathcal{L}$, with probability p'_m we drop label $l_{\nu,k}$ to simulate the case in which the ν-th AV missed to detect m_k, where the probability p'_m is a preset "probability of missed detection".

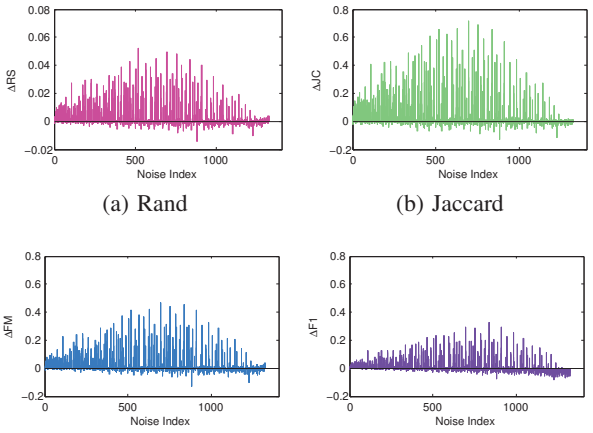
These two types of noise can affect, with different preselected probabilities, either one, two, or three AVs. To better explain this, let $\mathsf{n} = [p_f, p_m; p_1, p_2, p_3]$ be a "noise vector" whose elements express the following probabilities: p_f is the overall probability that a malware sample m will be affected by a label flip, while p_m is the overall probability that a sample will be affected by a missing label; on the other hand, p_x (with x =1,2, or 3) represents the probability



(a) Rand (b) Jaccard

(c) FM (d) F1

Figure 3: VAMO, absolute values of cluster validity indexes.



(a) Rand (b) Jaccard

(c) FM (d) F1

Figure 4: VAMO vs. Majority Voting (index "deltas").

that the noise (through label flips and/or missing labels) will affect exactly x out of the three AVs, for a given malware sample. Notice that $p_1 + p_2 + p_3 = 1$, and $p_f + p_m \leq 1$. Namely, with probability $1 - (p_f + p_m)$ a sample will not be affected by any noise (i.e., the sample remains perfectly labeled).

6.1.3 Building a Reference Clustering via Majority Voting

Intuitively, the majority voting-based approach to construct a reference clustering works as follows. Given a malware sample $m_i \in \mathbf{M}$, and the label vector $L_i = (l_{1,i}, l_{2,i}, l_{3,i}) \in \mathcal{L}_M$ containing the malware family labels assigned to m_i by the three AVs, m_i is assigned to malware cluster R_j if the majority of labels in L_i indicate that m_i belongs to family f_j. If no majority-based consensus can be reached (i.e., the majority of AVs disagree on the family name attributed to m_i), then the sample m_i is assigned to a *singleton cluster*, namely a cluster that contains only m_i. Following this approach, we can partition the dataset $\mathbf{M}$ into a set of majority voting-based reference clusters $\mathbf{Rc}^{MV} = \{Rc_1^{MV}, R_2^{MV}, ..., R_q^{MV}\}$. Then, given $\mathbf{Rc}^{MV}$ and the *ground truth* clusters $\mathbf{C} = \{C_1, ..., C_s\}$ (which are derived before injecting the noise into the AV labels), we can compute the four external validity indexes described in Section 3.2.

6.1.4 Computing the Validity Indexes

Let n be a particular noise vector, with a given combination of values for the probabilities p_f, p_m, p_1, p_2, and p_3. Applying the noise injection approach described above results in a noisy label dataset $\mathcal{L}(\mathsf{n})$. In turn, if from $\mathcal{L}(\mathsf{n})$ we only consider the labels related to the malware samples in $\mathbf{M}$, we can obtain a (noisy) label dataset $\mathcal{L}_M(\mathsf{n})$ (notice that because $\mathbf{M} \subset \mathcal{A}$, then $\mathcal{L}_M(\mathsf{n}) \subset \mathcal{L}(\mathsf{n})$).

Given $\mathcal{L}(\mathsf{n})$ and $\mathcal{L}_M(\mathsf{n})$, we apply VAMO to compute four validity indexes (see Figure 1), thus essentially measuring the *level of agreement* (see Section 4) between the reference clustering derived from the *AV Label Graph* learned from $\mathcal{L}(\mathsf{n})$, and the *ground truth* clusters $\mathbf{C} = \{C_1, C_2, ..., C_s\}$ in which $\mathbf{M}$ was originally partitioned (i.e., before any noise was applied). Let $RS^{VAMO}(\mathsf{n})$ be the resulting Rand statistic, $JC^{VAMO}(\mathsf{n})$ be the Jaccard coefficient, $FM^{VAMO}(\mathsf{n})$ be the Folkes-Mallows index, and $F1^{VAMO}(\mathsf{n})$ be the F1 index that combines precision and recall (see Section 3.2).

We similarly compute these four external cluster validity indexes by first applying the majority voting-based approach described in Section 6.1.3 over $\mathbf{M}(\mathsf{n})$ to obtain a reference clustering $\mathbf{Rc}^{MV}(\mathsf{n})$, and then comparing this reference clustering to $\mathbf{C}$. Let $RS^{MV}(\mathsf{n})$ be the resulting Rand statistic, $JC^{MV}(\mathsf{n})$ be the Jaccard coefficient, $FM^{MV}(\mathsf{n})$ be the Folkes-Mallows index, and $F1^{MV}(\mathsf{n})$ be the F1 index. Now, for each value of n we compute the difference between the validity indexes obtained using VAMO and the ones based on the majority voting approach. For example, we compute $\Delta RS(\mathsf{n}) = RS^{VAMO}(\mathsf{n}) - RS^{MV}(\mathsf{n})$, and in a similar way we also compute $\Delta JC(\mathsf{n})$, $\Delta FM(\mathsf{n})$, and $\Delta F1(\mathsf{n})$.

6.1.5 Results

Figure 3 reports the absolute values of the cluster validty indexes obtained using VAMO, while Figure 4 plots the difference between the four external validity indexes produced by the comparison between VAMO's results and the majority voting approach, as explained above. In Figure 3, the y axis reports the absolute value of the indexes, while in Figure 4 it reports the "deltas". In both cases, the x axis is simply the index of the experiment round, with the noise increasing per each experiment [5]. Specifically, we use 1,320 different noise configurations (i.e., different values of the elements of the noise vector n), with the only constraint that $p_f + p_m \leq 0.5$, i.e., at most 50% of the malware samples will be affected by some noise in their AV labels. It is also worth noting that the y axis for ΔRS varies in $[-0.02, 0.08]$, while all other "deltas" graphs have values on the y axis in $[-0.2, 0.8]$.

Figure 4(a) shows that the difference between RS^{VAMO} and RS^{MV} are relatively small, and ΔRS varies between -0.02 and 0.06. However, the three remaining validity indexes (Figure 4(b) through Figure 4(d)) clearly show that VAMO's reference clustering *agrees more closely* with the underlying true clustering $\mathbf{C}$, compared to the majority voting-based reference clustering. In fact, in all four indexes the "deltas" are positive for the vast majority of the noise combinations, meaning that the quality indexes obtained by VAMO show a better agreement with the true clustering, compared to the quality indexes obtained via majority voting.

To better analyze the effect of the noisy AV labels, Figure 5 through Figure 8 present the validity index "deltas" considering all noise vectors n for which: (a) at least two AVs are affected by noise, i.e., $p_1 = 0$; (b) the only type of noise affecting the labels is the "label flips", i.e., $p_m = 0$ (no missing labels, which means that all the AVs assign a malware family label to all samples); (c) the only type of noise is "missing labels", i.e., $p_f = 0$ (no label flips). As we can see, whenever the label noise (or inconsistencies) affects a majority of AVs (case (a)), or when any AV misses to detect some malware samples (case (c)), VAMO clearly outperforms the majority voting-based approach, because VAMO's reference clustering more closely agrees with the true malware clusters. While the "label flips" (i.e., case (b), which simulates the scenario in which AVs assign the incorrect malware family name) have a more negative effect on VAMO because they more heavily affect the edges (and their weights) learned through the *AV Label Graph*, VAMO performs comparably to majority voting, as shown by the very small negative "deltas".

[5]The experiment rounds are ordered according to a summary *noise level* computed as $nl = (0.6p_f + 0.4p_m) \cdot (0.1p_1 + 0.3p_2 + 0.6p_3)$.

l	*clusters*	Rand	Jaccard	Folkes-Mallows	F1
0.10	674	0.8767	0.2086	0.4494	0.7100
0.20	451	0.9172	0.5438	0.7308	0.7918
0.30	313	0.9205	0.5777	0.7482	0.7948
0.31	**301**	**0.9792**	**0.8924**	**0.9434**	*0.8436*
0.32	291	0.9790	0.8916	0.9430	0.8431
0.33	288	0.9759	0.8782	0.9357	**0.8501**
0.34	286	0.9759	0.8782	0.9357	0.8496
0.35	280	0.9758	0.8775	0.9353	0.8479
0.36	274	0.9757	0.8772	0.9352	0.8467
0.37	261	0.9721	0.8614	0.9265	0.8433
0.38	255	0.9721	0.8613	0.9265	0.8424
0.39	248	0.9722	0.8623	0.9270	0.8421
0.40	241	0.9721	0.8617	0.9268	0.8401
0.50	187	0.9585	0.8081	0.8971	0.7937
0.60	142	0.9260	0.7070	0.8366	0.7489
0.70	113	0.8527	0.5614	0.7354	0.7260
0.80	85	0.7789	0.4659	0.6656	0.7124

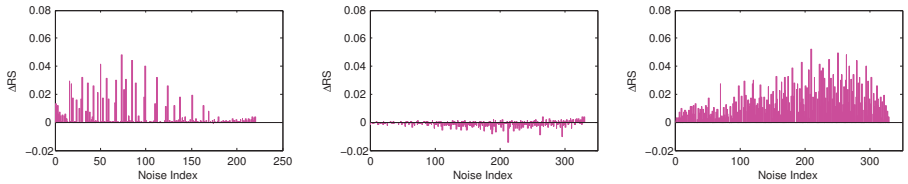
Table 2: Application of VAMO to behavior-based malware clustering results.

6.2 Real-World Application

In this Section, we discuss how VAMO can be applied in practice to assess the quality of the results produced by malware clustering systems. Specifically, we apply VAMO to the results that the behavior-based malware clustering system presented in [2] produced over a real-world malware dataset $\mathbf{M}$ containing 2,026 distinct malware samples collected in February 2009. To obtained the behavior-based clustering we proceeded as follows. We provided all malware samples in $\mathbf{M}$ to the authors of [2], who kindly agreed to analyze them and provide us a distance matrix $\mathbf{D}$ containing the pair-wise distances between the samples computed based on their system-level behavioral features. Given, $\mathbf{D}$, we applied *precise* average-linkage hierarchical clustering (this step is slightly different from [2], in which the authors applied an *approximate* hierarchical clustering algorithm), and obtained a dendrogram, which we will refer to as $\mathcal{Y}$ in the following. As usual, the dendrogram $\mathcal{Y}$ can be cut at a given height to obtain a partitioning of dataset $\mathbf{M}$ into a number of malware clusters (see discussion below).

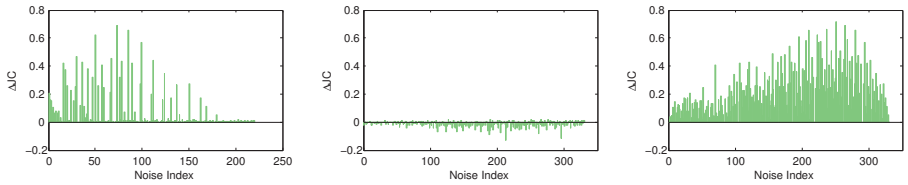
To generate VAMO's *AV Label Graph*, we used a dataset $\mathcal{A}$ consisting of 998,104 real-world distinct malware samples collected between August 2008 and August 2009. All of these 998,104 samples were scanned using four different popular AVs, in a way analogous to the malware dataset we discussed in Section 3.1, to obtain the label dataset $\mathcal{L}$. Each sample in this dataset was assigned at least one AV label. Also, $\mathcal{L}$ contained the labels for most of the 2,026 samples in $\mathbf{M}$. Specifically, $\mathcal{L}$ included at least one label for 1,985 samples in $\mathbf{M}$, while the remaining 41 samples were not represented in $\mathcal{L}$, and therefore remained *unlabeled*.

Taking the labeled dataset $\mathcal{A}$ and the labels for the samples in dataset $\mathbf{M}$ (including the placeholder "unknown" labels for the 41 samples that remained undetected) as input, we applied VAMO to produce a reference clustering, following the procedure outlined in



(a) F & M (> 1 AV) (b) Only Flips (c) Only Missing

Figure 5: VAMO vs. Maj. Voting: Rand Statistic.



(a) F & M (> 1 AV) (b) Only Flips (c) Only Missing

Figure 6: VAMO vs. Maj. Voting: Jaccard Coefficient.

Section 4 and Section 5. Then, given the dendrogram $\mathcal{Y}$ obtained from the third-party malware clustering system [2], we cut $\mathcal{Y}$ at different heights $l_1, l_2, .., l_n$, thus obtaining a sequence of different clusterings $\mathbf{C}(l_1), \mathbf{C}(l_2), .., \mathbf{C}(l_n)$. For each of these clustering results, we used VAMO to compute a set of validity indexes (see Section 5). Table 2 summarizes our results. The first column in Table 2 reports the value of the hight l at which $\mathcal{Y}$ is cut, while the second column reports the related number of clusters that was obtained from $\mathbf{M}$. For example, by cutting $\mathcal{Y}$ at height $l = 0.5$, $\mathbf{M}$ is partitioned into 187 clusters. The remaining columns represent the values of five different external cluster validity indexes (Section 3.2) measured by comparing the obtained malware clusters to VAMO's reference clustering, as explained in Section 5. We varied $l \in [0, 1]$ at steps equal to 0.01 (in practice, we excluded the extreme values $l = 0$ and $l > 0.8$, because they result either into one malware per cluster or into artificially large clusters, respectively). In the interest of space, because the maximum value of the validity indexes is located between $l = 0.3$ and $l = 0.4$, we report the results at steps of 0.01 only within that range.

As we can see from Table 2, the best value of the cut l is equal to 0.31, because that is the cut hight at which three out of four external validity indexes express the fact that there is maximum agreement between the behavior-based clusters and the AV labels generated by four different AV scanners. Put another way, VAMO's results indicate that the AV labels provide the best explanation of the underlying malware dataset $\mathbf{M}$ when $\mathbf{M}$ is partitioned into 301 clusters by cutting $\mathcal{Y}$ at $l = 0.31$.

It is worth noting that the F1 index is the only external validity index that is not maximum at $l = 0.31$. However, the value of 0.8436 obtained at $l = 0.31$ is quite close to the maximum value of 0.8502 reached at $l = 0.33$. This result suggests that to find the best configuration parameters for the third-party malware clustering system, it may be better to consider multiple validity indexes, rather than focusing only on analyzing precision and recall (and the related F1 index), as proposed in previous work [2, 11].

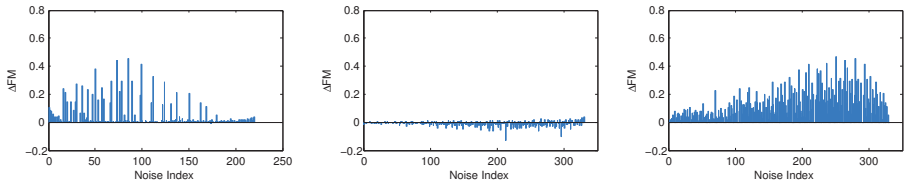
7. DISCUSSION

Using AV labels to build a reference clustering has some potential limitations, even though the label inconsistencies can be mitigated using VAMO. First, we need to take into account that the features used by the AVs to characterize malware samples and assign them to a given malware family may be different from the features used by a third-party malware clustering system to measure the similarity among samples. For example, AV vendors often base their malware categorization process on features extracted from reverse engineering the malware binaries. On the other hand, behavior-based malware clustering systems leverage features related to the malware's system [2] or network activities [15], for example. Naturally, different features may highlight different types of similarities in the samples. Therefore, while the AV labels clearly represent a valuable point of reference, especially in absence of a more perfect ground truth, the comparison between behavior-based malware clustering results and AV family labels should be taken with a grain of salt. A similar argument is made in [4], in which the authors outline the potential pitfalls of using labeled datasets meant for training and testing of supervised learning algorithms for evaluating the effectiveness of (unsupervised) clustering algorithms. Nonetheless, AV label-based cluster validity analysis, especially when fully automated such as in VAMO, is certainly a valuable tool that can assist malware analysts in the analysis of their malaware clustering results.

Another factor to consider is the fact that AV labels *evolve* in time. That is, a malware sample m assigned by an AV to family f_i at time t_0, may be "renamed" by the same AV as belonging to a different family f_j at a future time $t_1 > t_0$. This is due to the fact that AV signatures are sometimes refined by the AV vendors to reduce possible false positives and more specifically characterize the malware samples (e.g., by assigning a sample previously labeled as "generic" to a more specific malware family). To take this into account, the historic archive of malware labels used by VAMO should be kept updated. This may be done by either periodically rescanning the malware dataset, or by querying online services such as `virustotal.com`.

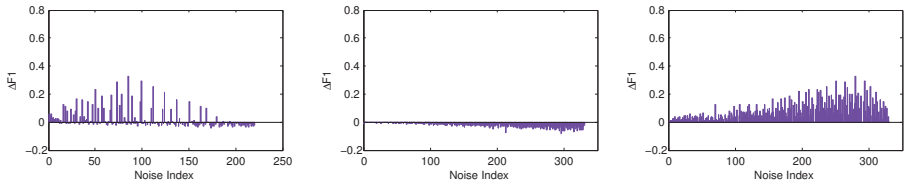
8. CONCLUSION

In this paper, we presented a novel system, called VAMO, that provides a fully automated assessment of the quality of malware clustering results. Previous studies propose to evaluate malware clustering results by leveraging the labels assigned to the malware samples by multiple AVs. However, they require a manual mapping



(a) F & M (> 1 AV) (b) Only Flips (c) Only Missing

Figure 7: VAMO vs. Maj. Voting: Folkes-Mallows.



(a) F & M (> 1 AV) (b) Only Flips (c) Only Missing

Figure 8: VAMO vs. Maj. Voting: F1 Index.

between labels assigned by different AV vendors, and are limited to selecting a *reference sub-set* of samples for which an agreement regarding their labels can be reached across a majority of AVs.

Unlike previous work, VAMO does not require a manual mapping between malware family labels output by different AV scanners. Furthermore, VAMO does not discard malware samples for which a majority voting-based consensus cannot be reached. Instead, VAMO explicitly deals with the inconsistencies typical of multiple AV labels to build a more representative reference set. Our evaluation, which includes extensive experiments in a controlled setting and a real-world application, show that VAMO performs better then majority voting-based approaches, and provides a way for malware analysts to automatically assess the quality of their malware clustering results.

Acknowledgments

This material is based in part upon work supported by the National Science Foundation under Grant No. CNS-1149051. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario. Automated classification and analysis of internet malware. In *Recent Advances in Intrusion Detection*, 2007.

[2] U. Bayer, P. Milani Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, behavior-based malware clustering. In *Network and Distributed System Security Symposium*, 2009.

[3] M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. In *ACM SIGSOFT symposium on the foundations of software engineering*, ESEC-FSE '07, 2007.

[4] I. Färber, S. Günnemann, H. Kriegel, P. Kröger, E. Müller, E. Schubert, T. Seidl, and A. Zimek. On using class-labels in evaluation of clusterings. In *MultiClust: 1st International Workshop on Discovering, Summarizing and Using Multiple Clusterings Held in Conjunction with KDD*, 2010.

[5] E. B. Fowlkes and C. L. Mallows. A method for comparing two hierarchical clusterings. *Journal of the American Statistical Association*, 78(383):553–569, 1983.

[6] F. Guo, P. Ferrie, and T. Chiueh. A study of the packer problem and its solutions. In *Recent Advances in Intrusion Detection*, 2008.

[7] M. Halkidi, Y. Batistakis, and M. Vazirgiannis. On clustering validation techniques. *J. Intell. Inf. Syst.*, 17(2-3):107–145, 2001.

[8] X. Hu, T.-c. Chiueh, and K. G. Shin. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, 2009.

[9] A. K. Jain and R. C. Dubes. *Algorithms for clustering data*. Prentice-Hall, Inc., 1988.

[10] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Comput. Surv.*, 31(3):264–323, 1999.

[11] J. Jang, D. Brumley, and S. Venkataraman. Bitshred: feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, 2011.

[12] P. Li, L. Liu, D. Gao, and M. K. Reiter. On challenges in evaluating malware clustering. In *Proceedings of the 13th international conference on Recent advances in intrusion detection*, RAID'10, 2010.

[13] F. Maggi, A. Bellini, G. Salvaneschi, and S. Zanero. Finding non-trivial malware naming inconsistencies. In *International Conference on Information Systems Security*, ICISS'11, 2011.

[14] M. Meilă. Comparing clusterings—an information based distance. *J. Multivar. Anal.*, 98(5):873–895, May 2007.

[15] R. Perdisci, W. Lee, and N. Feamster. Behavioral clustering of http-based malware and signature generation using malicious network traces. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'10, 2010.

[16] D. Pfitzner, R. Leibbrandt, and D. Powers. Characterization and evaluation of similarity measures for pairs of clusterings. *Knowl. Inf. Syst.*, 19(3):361–394, May 2009.

[17] E. RendŮn, I. Abundez, A. Arizmendi, and E. M. Quiroz. Internal versus external cluster validation indexes. *universitypressorguk*, 5(1), 2011.

[18] K. Rieck, P. Trinius, C. Willems, and T. Holz. Automatic analysis of malware behavior using machine learning. *J. Comput. Secur.*, 19(4):639–668, Dec. 2011.

[19] Symantec. Symantec internet security threat report, trends for 2010, April 2011.

Towards Network Containment in Malware Analysis Systems

Mariano Graziano
Institut Eurecom
graziano@eurecom.fr

Corrado Leita
Symantec Research Labs
corrado_leita
@symantec.com

Davide Balzarotti
Institut Eurecom
balzarotti@eurecom.fr

ABSTRACT

This paper focuses on the containment and control of the network interaction generated by malware samples in dynamic analysis environments. A currently unsolved problem consists in the existing dependency between the execution of a malware sample and a number of external hosts (e.g. C&C servers). This dependency affects the repeatability of the analysis, since the state of these external hosts influences the malware execution but it is outside the control of the sandbox. This problem is also important from a containment point of view, because the network traffic generated by a malware sample is potentially of malicious nature and, therefore, it should not be allowed to reach external targets.

The approach proposed in this paper addresses the repeatability and the containment of malware execution by exploring the use of protocol learning techniques for the emulation of the external network environment required by malware samples. We show that protocol learning techniques, if properly used and configured, can be successfully used to handle the network interaction required by malware. We present our solution, Mozzie, and show its ability to autonomously learn the network interaction associated to recent malware samples without requiring a-priori knowledge of the protocol characteristics. Therefore, our system can be used for the contained and repeatable analysis of unknown samples that rely on custom protocols for their communication with external hosts.

Keywords

Malware containment, protocol learning, network traffic replay

1. INTRODUCTION

Dynamic analysis is a useful instrument for the characterization of the behavior of malware samples. The most popular approach to perform dynamic analysis consists in the deployment of *sandboxes*, i.e., instrumented environments in which a malware sample is run, and in which detailed information on the actions performed by the sample is logged. A variety of different approaches has been explored for the collection of host-based information in sandboxed environments. These range from the API hooking approaches adopted by CWSandbox [4] and the Cuckoo sandbox [6], as well as more elegant and less intrusive techniques such as the ones used by TTAnalyze and Anubis [8, 3]. The output of the sandbox analysis has proven to be of extremely useful to malware analysts, both to study the execution of a single sample and to identify behavioral commonalities among different samples [9].

However, the result of the execution of a malware sample in a sandbox is highly dependent on the sample interaction with other Internet hosts. This was clearly described by Rossow et al. in [24], where the authors analyzed the interaction between malware and Internet hosts during extended execution in a sandboxed environment. The study underlined the critical dependance on remote hosts for the download of additional malware components and for the C&C coordination.

The network traffic generated by a malware sample also raises obvious concerns with respect to the containment of the malicious activity. In fact, it is important to ensure that the correct execution of the sample does not cause any damage to other external hosts, for instance in the context of self-propagation attempts. However, these concerns go beyond the containment issue and relate directly to the quality of the analysis itself. For instance, Leita et al. [17] compared the output of the clustering algorithm presented in [9] with other information sources, such as static malware clustering techniques and honeypot data. This empirical study clearly showed how polymorphic variants of the same malware sample could be erroneously associated to different groups depending on the state of their C&C server at the moment in which the samples were executed.

In this paper we address two problems. The first is the poor repeatability of malware analysis experiments. Malware analysts often execute samples inside a sandbox, in order to observe and collect their malicious behavior. However, the exhibited behavior may depend on external factors, such as the commands received by a C&C server or the content of a given URL. For example, consider a classic scenario common to many security companies. Collected samples are automatically analyzed by a malware analysis system, and their behavior (e.g., filesystem operations, process creations, and modification to the Windows registry) is stored in a database. When a program requires a closer look, an analyst can run it again in a separate, better in-

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

strumented environment, for example by using a debugger or by collecting all the system calls. Unfortunately, it is often the case that these "secondary inspections" are performed several days, or even weeks after the samples were initially collected, with the risk of studying dead samples for which the required infrastructure is not available anymore. In fact, the remote machines contacted by malware are volatile by nature, often hosted on other compromised computers, or taken down by providers and law enforcement when the malicious activity is detected. Therefore, the malware infrastructure required to properly run the sample is normally available for only a limited amount of time.

The second problem we address in this paper is related to malware containment, i.e., to the ability of properly execute a given sample in an isolated environment, where it cannot cause any harm to the rest of the world. In general, full containment of a new, previously unknown, sample is impossible. However, we can identify two scenarios in which such result could be achieved: the execution of a polymorphic variation of an already analyzed malware, and the re-execution of a previously studied sample. In both cases, the sample (or a behaviorally equivalent variation of it) has already been analyzed by the system and, therefore, the problem of full containment can be reduced to the previous problem of repeatable execution. The idea is that if we can "mimic" the behavior of the network to match the one observed in a previous execution, we can obtain at the same time a repeatable experiment and a containment of the malware execution.

To mitigate these issues, in this paper we want to study to which extent it is possible to enrich the information collected during malware execution to make the experiments repeatable and achieve full network containment. In particular, we experiment with a protocol-agnostic technique [20] previously adopted to model the attack traffic in high-interaction honeypots [18, 16]. The idea consists in building a finite state machine (FSM) of the network activity generated by each malware sample. The extracted FSM can be stored alongside the other collected information, and can then be used to properly "simulate" all the required endpoints whenever the sample needs to be analyzed again in the future.

It is important to note that in this paper we only address *network* repeatability: Our goal is to ensure that the malware finds all the remote components it needs to properly execute. On the contrary, we do not address full process repeatability, i.e., the problem of forcing two executions of a malicious executable to behave exactly the same from an operating system point of view. Balzarotti et al. [7] studied the problem of full repeatability in the context of the detection of split personalities in malware. Their prototype works at the system-call level and has several technical limitations, making it hard to deploy on current malware sandboxes.

To summarize, the paper makes the following contributions:

- We discuss how protocol learning techniques can be used to model the traffic generated during the execution of malware samples. In particular, we describe the limitations of protocol-agnostic approaches and show that, if properly setup and configured, they can be used to successfully replay real malware conversations.
- We describe the implementation of Mozzie, a network containment system that can be easily applied to all the existing sandbox environments. According to our experiments, an average of 14 network traces are required by Mozzie to model the traffic and achieve full containment for real malware samples.

Approach	Containment	Quality
Full Internet access	×	∼
Filter/redirect specific ports	∼	∼
Common service emulation	√	∼
Full isolation	√	×

Table 1: Network access strategies in dynamic analysis

The rest of this paper is structured as follows. In Section 2 we introduce the problem of network repeatability and containment and we discuss related work on the topic. In Section 3 we briefly describe the core ideas underneath the protocol learning algorithms adopted in this paper. In Section 4 we present our approach, the architecture of our system, and the details of our prototype implementatoin of Mozzie. We then present and discuss the results of our experiments in Section 5, and and the limitations of our approach in Section 6. Finally we conclude the paper in Section 7.

2. MALWARE ANALYSIS AND CONTAINMENT

Several different strategies have been proposed in the state of the art to address the problem of network containmnent and the quality of the dynamic analysis. In particular, the concept of quality refers to both the need to allow connectivity to external hosts (to expose the malware interesting behavior) and to the need to make the analysis process repeatable. Table 1 summarizes the previous work in four different categories.

Full Internet access. The most straightforward approach consists in providing the sandbox with full Internet access. A similar approach is however unacceptable from a containment standpoint: the malware sample is left free to propagate to victims, or to participate to other types of malicious activities (DoS, spam, ...). The quality of the analysis is also only partially acceptable: the sample is left free to interact with external hosts upon execution, but its behavior becomes dependent on the state of external hosts, leading to the problems underlined in [17].

Filter/redirect specific ports. The containment problem associated to Full Internet access is rarely discussed in Internet-connected sandboxes such as Anubis [3], CWSandbox [4] and others. From informal discussions with the maintainers, it appears to be common practice for the public deployment of these sandboxes to employ simple filtering or redirection rules, in which TCP ports commonly associated to malicious scans (e.g. port 139 and port 445) are either blocked or redirected towards honeypots. This partially solves the containment problem: SMB vulnerabilities are a very common propagation vector for self-propagating

malware, that can be easily prevented with such measures. However, this approach is not able to deal with other types of activity whose nature cannot be easily discerned from the TCP destination port. A similar attempt to perform containment through redirection was implemented also in the context of honeyfarms such as Potemkin [28] and GQ [12]. In such deployments, the authors had investigated the idea of reflecting outbound traffic generated by infected virtual instances of the honeyfarm towards other instances of the same honeyfarm. A similar approach proved to be valuable for the analysis of malware propagation strategies, but was not effective at dealing with other types of traffic such as C&C communication. In fact, redirecting a C&C connection attempt towards a generic honeyfarm virtual machine is not likely to generate meaningful results. Kreibich et al. [15] have recently improved GQ making it a real and versatile malware farm. They have addressed the containment problem with precise policies but their approach has not dealt with the repeatability issue.

Common service emulation. Sandboxes such as Norman Sandbox prevent the executed malware from connecting to the Internet, and provide instead generic service implementations for common protocols such as HTTP, FTP, SMTP, DNS and IRC. A similar approach was revisited and enhanced by Ionue et al. in [14], a two-pass malware analysis technique in which the malware sample is allowed to interact with a "miniature network" generated by an Internet emulator able to provide a variety of dummy services to the executed malware sample. All these approaches are however limited, and rely on a-priori knowledge of the communication protocols employed by the malware sample. Malware often uses variations of standard protocols, or completely ad-hoc communication protocols that cannot be handled through dummy services. Yoshioka et al. [30] have tried to tackle this problem by incrementally refining the containment rules according to the dynamic analysis results. While such an approach provided an elegant solution to the containment problem, it did not address the quality of the analysis and it did not attempt to remove dependencies between the malware behavior and the state of the external Internet hosts involved in the analysis.

Full isolation. Completely preventing the malware sample from interacting with Internent hosts ensures a perfect containment of its malicious activity. However, the complete inability to interact with C&C servers and repositories of additional components is likely to severely bias the outcomes of the dynamic analysis process.

Table 1 underlines a partial trade-off between the containment problem and that of ensuring the quality and repeatability of the analysis. On the one hand, running the malware sample in full emulation addresses all the containment concerns but, by barring the malware sample from communicating with the external hosts it depends on, it strongly biases the results of the dynamic analysis (i.e., the sample may only go as far as trying to connect to the hosts but without exposing any real malicious behavior). On the other hand, providing the sandbox with full Internet connectivity increases the analysis quality but it does not solve the repeatability problem, and it also raises important ethical and legal concerns.

This paper aims at addressing this problem by exploring the use of protocol learning techniques to automatically create network interaction models for the hosts the malware depends on upon execution (even in presence of custom and undocumented protocols), and using such models to provide the sandbox with an isolated, yet rich network environment.

3. PROTOCOL INFERENCE

A common problem when looking at the network interaction generated by a malware sample is associated to the interpretation of application-level protocols. Malware samples often propagate by exploiting vulnerabilities in poorly documented protocols (such as SMB), and rely on custom protocols for their coordination with the C&C servers [26, 25, 27]. The execution of the sample in isolation requires us to trick the malware sample in interacting with replicas of the real Internet hosts the malware interacts with (victims, C&C servers, ...), but without being able to assume a-priori knowledge of the application-level protocol implemented in such services.

This challenge is addressed in this paper by resorting to protocol learning techniques. Techniques such as ScriptGen [20, 19], RolePlayer [13], Discoverer [11] and Netzob [5] aim at partially reconstructing the protocol message syntax and, in some cases, the protocol Finite State Automata [19, 5] from network interactions. The inference is performed by looking at network traces generated by clients and servers while minimizing the number of assumptions on the protocol characteristics. Differently from other approaches such as [10, 22, 21, 29], that factor into the protocol analysis also host-based information obtained through execution monitoring or memory tainting, the above methods focus on the extraction of the protocol format solely from network traces and are therefore particularly suitable to our task, where we are unable to control the remote endpoints contacted by malware.

While any of the previouly mentioned tools would be suitable for this task, in this paper we focus on ScriptGen [20, 19] because of its limited amount of assumptions on the protocol characteristics and its support to the inference not only of single protocol messages, but also of their structure within the protocol flow. ScriptGen is an algorithm that generates an approximation of a protocol language by means of a "server-side" Finite State Machine whose scope corresponds to a specific TCP connection or UDP flow: the FSM root corresponds to the establishment of the connection, while any of the FSM leaves corresponds to the point in which the connection is successfully closed. In the FSM representation each transition is labelled with a regular expression matching a possible client request, while each state is labelled with the server answer to be sent back to the client when reaching that specific state.

ScriptGen performs this task through two subsequent processes, as illustrated in Figure 1:

1. **Semantic clustering.** Initially introduced in [19], the semantic clustering algorithm aims at grouping together protocol messages likely to be semantically similar. Any new conversation to be added to an ex-

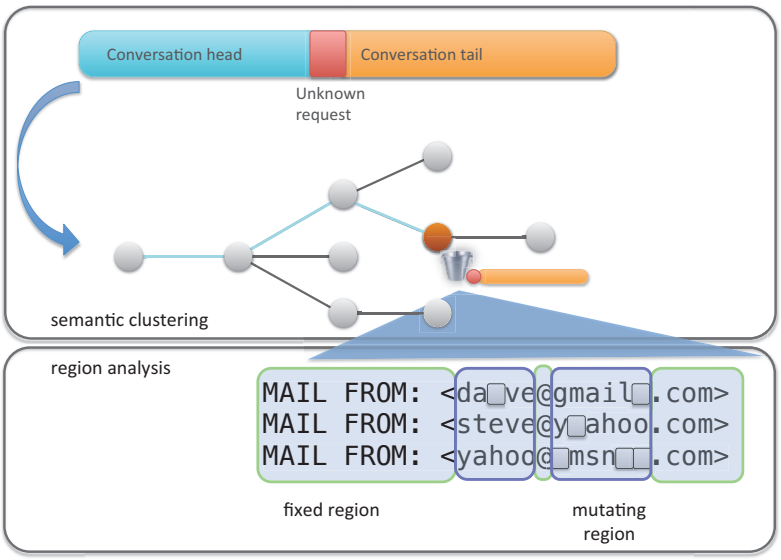


Figure 1: Simplified diagram of the ScriptGen operation

isting protocol model will consist of two parts: an initial part (*head*) composed of messages already modeled in the current version of the FSM, and a final part (*tail*) composed of messages that do not match the current model. A 0-length head represents a conversation whose first message already differs from the current protocol model, while a 0-length tail models a conversation that already fully matches the existing protocol knowledge. Semantic clustering aims at grouping together messages sharing the same head (by matching the head messages against the current FSM model) and likely to be sharing the same tail (by clustering conversations sharing the same head according to the length of the tail messages).

2. **Region analysis.** Semantic clustering leads to the identification of protocol messages that are very likely to be associated to a specific semantic context. The process that generalizes the specific protocol messages into a set of regular expressions able to correctly recognize future messages sharing the same semantic value is called region analysis and was introduced in [20]. Through the analysis of multiple versions of a semantically similar message, the region analysis algorithm aims at identifying *regions* with similar characteristics. Through the subsequent application of global alignment algorithms (Needleman-Wunsch [23]), ScriptGen refines the clustering of the messages separating those exposing major structural differences (macroclustering) while correctly dealing with variable length fields. The final result consists in a global alignment of each identified group. The statistical analysis of each byte of the alignment outcome is used to identify protocol *regions*, i.e., portions of protocol message likely to have a specific semantic value. The distribution of the content of each region over the different samples gives us information on their semantic nature: regions with random content throughout the set of samples are likely to be nonce values, regions with fixed content are likely to be separators or semantically reach protocol fields, and regions mutating through a limited amount of possible values are likely to be associated to more subtle protocol semantics, whose preservation is decided through a *microclustering* threshold. The final outcome of this semantic evaluation is a set of regular expressions, each of which will lead to the generation of a new subtree in the FSM object.

Most of the ScriptGen operation is driven by thresholds, that regulate for instance the different clustering steps. A simple approach for tuning the thresholds to the best configuration was introduced in [20] and consisted in brute-forcing all the possible combinations of thresholds to identify the global optimum. While this is an computationally expensive process, the computed thresholds proved to be sufficiently robust to handle protocols with "similar" characteristics in terms of amount of variability in their structure.

It is important to understand that ScriptGen avoids any assumption on the nature of the protocol separators or on the possible representation of semantically relevant fields, and performs a partial reconstruction of the protocol semantics by analyzing at the same time multiple samples of the same type of protocol exchange (*conversation*). The higher the number of conversations available to the algorithm, the more precise the protocol inference process will be. Intuitively, if we consider two message instances of a protocol containing a random cookie value, their cookie value could be *aabd* and *awed*. ScriptGen would have no way to consider such protocol section as a mutating region of 4 characters, since (by accident) both values start with 'a' and end with 'd'. These accidental false inferences can be filtered out only by considering a sufficiently large number of conversation samples.

ScriptGen has been successfully used in the past to automatically generate models of network behavior for the emulation of vulnerable services in exploits. This emulation was included in a distributed honeypot [16, 18] deployment, in order to emulate 0-day exploits and collect information

on the propagation vector of self-propagating malware. This paper investigates the use of ScriptGen in a different context, that of malware analysis, and tries to leverage its properties to automatically generate network models for the remote endpoints involved in the execution of a malware sample in a sandboxed environment.

4. SYSTEM OVERVIEW

Our approach to achieve repeatability and containment in malware analysis experiments can be summarized in four steps:

1. **Traffic Collection** - In this phase the system collects a number of network traces associated with the execution of a certain malware sample. This can be done by running the malware in a sandbox, or by extracting existing traces generated by past analyses.

2. **Endpoint Analysis** - This is a cleaning and normalization process applied to all the collected traces. Its main goal consists in removing anomalous traces that could affect the results and the associated conclusions. Each trace is then normalized to remove endpoint randomization, such as the one introduced by IP fluxing techniques.

3. **Traffic Modeling** - This phase aims at the automated generation of models starting from the collected traffic samples and at their subsequent storage in a compact representation. The modeling can be performed in two different ways: in an online fashion (from now on called *incremental learning*), in which the model is initially very simple and it is subsequently refined at every new execution of the sample, or in an *offline* fashion, more suitable for the analysis of previously collected network dumps. The actual logic used to model the traffic is implemented in a separate component of our system. As we already explained in Section 3, the current implementation is based on the ScriptGen approach, but other unsupervised algorithms can be easily plugged into our system to achieve the same result.

4. **Traffic Containment** - In this last phase, we use the model extracted in the previous step to mimic the network environment required by the malware sample. The containment system is implemented as a transparent proxy. When the model is sufficiently precise, the proxy is able to mimic the external world, effectively achieving "full containment". When the model is incomplete, the proxy redirects the requests it cannot handle to the real targets. In this case, the system also collects the forwarded traffic to improve the training set, effectively closing the loop back to Step 1.

In the rest of the section, we introduce each phase in details, and we describe how each of them have been implemented in our system.

4.1 Traffic Collection

Collecting the malware traffic is as simple as running a network sniffer while the sample is running in the sandbox. For instance, several online systems (e.g., Anubis [3], and CWSandbox [4]) allow users to download the pcap file recorded during the analysis phase.

As explained in Section 5, we used two different datasets for our experiments, one extracted from old Anubis reports, and one collected live by running the samples inside a Cuckoo's sandbox. Finally, in order to be consistent with the data collected in the past, also in our experiments we limited the malware analysis and the network collection time to five minutes per sample.

4.2 Endpoint Analysis

The second phase of our process consists in cleaning and normalizing the collected traffic to remove spurious traces and improve the effectiveness of the protocol learning phase when facing network-level randomization.

The cleaning phase mainly consists in grouping together traces that exhibit a comparable network behavior. The intuition underneath this cleaning process is that the traces may have been generated at different points in time, and may capture different "states" of the remote endpoints. Most of the traces are likely to have been generated when the malware was indeed fully active, but we may still have to deal with a minority of traces that may have been generated, for example, when the C&C server was temporarily not reachable. It should be noted that the semantic clustering process explained in Section 3 allows ScriptGen to correctly deal with these cases. However, in this paper we are interested in evaluating the efficiency of our method at correctly leveraging *useful* traces to generate usable models, and thus we choose to clean the dataset from these spurious traces. In practice, this is achieved by clustering together traces according to each involved destination endpoint, where endpoint is defined as an $(IP, port)$ tuple. We consider the cluster with the highest amount of traces having similar high-level network behavior as the one representing the state of interest for our experiments.

While the cleaning process succeeds in most of the cases, in some of our experiments we noted that the clustering algorithm failed to identify a predominant network behavior for a specific sample. Closer investigation revealed that this failure is associated to the introduction of randomization in the network behavior of the sample. The most common example of this phenomenon is associated to malware using IP fluxing techniques. IP flux, also known as fast-flux, is a DNS-based technology used by malware writers to improve the resilience of their (often botnet) architecture. The idea is to rapidly swap the IP address associated to a particular domain name, to avoid having a single-point of failure and reducing the effect of IP blacklisting.

When a malware uses IP fluxing, most of the collected network traces will involve different endpoints. In other words, instead of having ten samples of conversations associated to a single endpoint, we will have ten different targets associated to one conversatione each. As we will see in the discussion of the Traffic Modeling component, this situation plays against our choice of creating protocol models on a per-endpoint basis: each endpoint will not be associated to a sufficient amount of samples to generate a meaningful model. However, this phenomenon is easy to detect because our endpoint clustering component would return an unusual result composed of many clusters containing a single trace each. In this case we automatically "normalize" the endpoints by identifying the DNS request that returned different IP addresses and by forcing it to return always the same value. In these cases, our system automatically replaces each

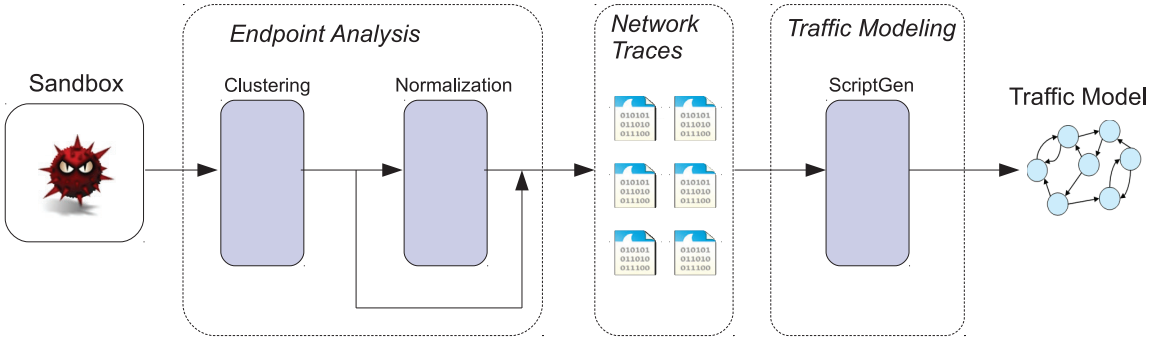


Figure 2: Creation of a Traffic Model

fluxed IP with the normalized IP in all the subsequent network flows. By performing this simple step we can obtain an uniform learning dataset, ready to be analyzed by the next stage of our system.

4.3 Traffic Modeling

Starting from the set of network traces obtained from the previous phase, the Traffic Modeling phase leverages the protocol learning algorithm (in our case, ScriptGen) to generate models for the network interaction with each endpoint involved in the malware execution. The models are maintained in the form of a dictionary, where each encoutered destination IP address and destination port is associated to its corresponding model (see Figure 2). This dictionary is later used to mimic the whole network environment, with the goal of containing all the requests generated by the malware.

4.4 Containment Phase

The goal of the containment phase is to "trick" the malware sample into believing that it is connected to the Internet, while in fact all the traffic is artificially generated by leveraging the information contained in the current protocol model.

The main component of the containment phase is the FSM player. The player is responsible for analyzing the incoming packets looking for new connections attempts, and checking if the contacted endpoint is present in the dictionary of FSMs. If so, the corresponding model is loaded and associated to the connection. Then, for each message, the player analyzes the current state and computes the next state. This is done by finding the best transition that matches the incoming content and by extracting the corresponding list of possible answers.

Figure 3 shows the operation of our system over two possible operation modes: *full* and *partial containment*.

In the *full containment* case, the protocol model is accurate enough to allow our system to correctly generate a response for every network interaction generated by the malware upon execution.

In the *partial containment* case, instead, the protocol model is inaccurate or incomplete: some of the network interactions created by the malware are not modelled in the associated FSM. Whenever a message cannot be matched with the current FSM, the system is unable to further emulate the associated remote endpoint. In such case, the system enters in replay mode for that specific endpoint, and replays all the traffic generated so far towards the real Internet host associated to it. By replaying all the traffic generated so far, we are able to handle possible authentication steps that the malware may have already performed in its interaction with the FSM. The answer to the unknown request is eventually delivered by the real endpoint, and all the subsequent interaction is then relayed by the system (proxy mode).

This process needs to be carried out without affecting the malware execution. For instance, consider the example in Figure 4 in which the malware sends three messages $\{Msg_1, Msg_2, Msg_3\}$ over a TCP connection. The player is able to follow the first two messages on the FSM, thus returning the corresponding responses $\{Resp_1, Resp_2\}$. However, the third message is different from what it was expecting, and it does not know what to answer. Therefore, Mozzie opens a new TCP connection to the original target, and quickly replay the messages 1 and 2 in chronological order to bring the new connection to the same state of the one it is simulating with the malware. Then it sends the Msg_3 and switches to proxy mode. In proxy mode, the system acts like a transparent proxy, forwarding each packet back and forth from the malware to the endpoint on the Internet. When the connection is terminated, the data is used to incrementally improve the model, so that it could handle the same conversation in the future.

By executing a malware sample multiple times, we are therefore able to gradually and automatically move from partial containment (in case in which part of the generated interacton is still unknown) to full containment, where all the malware network behavior is fully modeled and emulated by the system. However, it should be noted that the full learning of the network behavior of the malware is different from the full knowledge of the protocol. It is possible to follow a malware without contacting the external server in all its executions within the sandbox, but this does not mean all the commands of the protocols have been discovered.

4.5 System Implementation

In the previous sections we explained the architecture of the system. Now we can describe how Mozzie, our prototype implementation, is realized.

Mozzie is based on *iptables*. In particular it uses the NFQUEUE userspace packet handler with the Python nfqueue-bindings [2]. This allows our user-space component to accept, drop, or modify each incoming packet. To decode the packets that are in the queue, we used the Scapy [1] library.

Our current prototype handles three different IP proto-

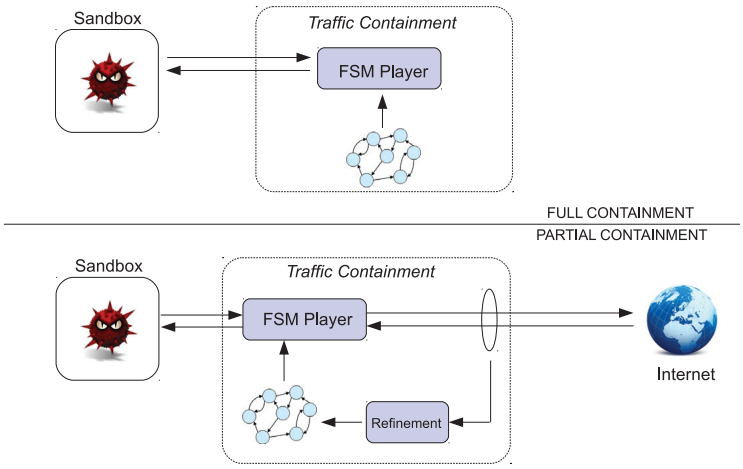


Figure 3: Replaying Architecture

cols: ICMP, UDP and TCP. ScriptGen cannot model the ICMP protocol, because it lacks the concept of *port* that is required to build the Finite State Machine. Therefore, Mozzie intercepts all the ICMP ECHO request messages and always answers with an ECHO reply. Beside that, the system mainly works as a userspace NAT, changing the destination IP and port for each TCP or UDP packet, to redirect them to the emulator responsible for that endpoint. The emulator runs an implementation of the ScriptGen algorithm and one instance of the FSM Player for each endpoint contacted by the malware. For example, if the malware opens a new TCP connection toward $(IP, Port)$, Mozzie checks if a FSM exists for that endpoint. If it finds one, it starts one FSM Player process to handle the connection and start redirecting the packets toward it. If not, it let the packets pass through so they can reach the real destination on the Internet.

Finally, the endpoint analysis is implemented as a series of Python scripts. One is responsible to process the available network traces and to cluster them together according to the contacted endpoints. The normalization is implemented by a separate tool that dissects the packets, changes the answer of the DNS requests, and replaces the corresponding IPs in the rest of the network traffic.

5. EVALUATION

In this section we describe the experiments we performed to evaluate Mozzie's ability to model real malware traffic. All the experiments were performed on an Ubuntu 10.10 machine running ScriptGen, Mozzie, and iptables v1.4.4. To perform the live experiments, we ran all samples in a Cuckoo Sandbox [6] running a Windows XP SP3 virtual machine.

5.1 System Setup

The goal of our evaluation is to automatically find the minimum number of network traces required to generate a finite state machine that can be used to fully contain the network traffic generated by a given malware sample.

To reach this goal, the first step of our experiments consisted in testing ScriptGen and properly tuning its parameters for the protocols we wanted to model. In fact, in our system we use ScriptGen to model the network behavior of

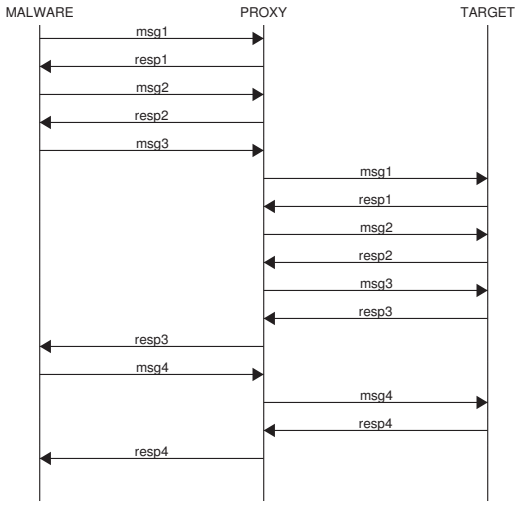


Figure 4: Sequence of messages during traffic replay

a generic program, but this is not the scenario for which the learning protocol tool was designed in the first place. As we already explained in Section 3, the best thresholds of ScriptGen's learning algorithm were experimentally set to the values that were observed to work well (in average) for network worms and remote exploits. However, those thresholds need to be re-computed for different protocols, in particular when moving from a text-based (e.g., HTTP) to a binary format (e.g., RPC).

In the first part of our experiments, we performed a number of tests to learn the optimal parameters for a number of protocols that are commonly used by several malware samples, namely HTTP, IRC, DNS, and SMTP. This step requires the algorithm to be executed several times on the same protocol traces, each time with different parameters [1]. Once the optimal setup was reached, we reused the same values for all the malware samples that used the same protocol. However, in one of the experiments, we discovered that the

[1]Please refer to [20, 19] for a description of the procedure required to set the thresholds

malware under analysis implemented a custom binary protocol. Since we did not have the ScriptGen configuration for that protocol, we had to re-apply the learning phase for that particular sample traffic.

Even though this operation can take several hours, it is important to note that if the thresholds are not set to their optimal values our technique would still be able to model unknown protocols, even though the system would require an higher number of traces to reach the full containment.

5.2 Experiments

Our experiments with real malware can be divided in two groups. In the first case, that we label *"offline"* learning, we use our system to model old traces collected in the past for polymorphic samples. Malware sandboxes normally avoid executing the same sample multiple times, returning the previously computed results when they recognize (usually from the MD5) that a file was already analyzed in the past. However, in case of polymorphic variations, it is possible that the same malware family gets executed several times. Based on this observation, we extracted from the Anubis [3] database the network traffic dumps associated to five polymorphic samples that adopted cleartext protocols in their communication.

Our goal was to show that, by using these traces, we can model the network behavior of the malware, and use the extracted FSM to replay and contain the execution of any other polymorphic variation of the same sample.

The results of this first experiment are reported in Table 2. The first two columns report the antivirus label and the malware category associated to each sample. The next column reports the success of the experiment, where a `FULL` (100%) value means that full containment was achieved and all the packets were properly replayed. In only one case, for the Koobface malware, our system was not able to successfully model the entire network traffic. The reason is the fast flux approach adopted by Koobface in which both the domain names *and* the IP addresses rotate. This makes it impossible for Mozzie to correctly model the DNS protocol, since there are not two sequences of request/response that look the same in the dataset.

Column 4 shows whether the normalization step was applied to the traffic. As we already explained in Section 4, clustering is required to deal with noise in the network traces, most of the time introduced by an anomalous execution of the malware (e.g., due to a network timeout on a web request). On top of that, certain malwares require a normalization phase to properly sanitize the traces from randomization introduced by IP or domain flux techniques.

Finally, the last column of the table shows the number of input traces required to successfully model the traffic. We started the experiment by running Mozzie on a single network trace. We then loaded the extracted model in a virtual machine and used it to try to contain five consecutive executions of another polymorphic variation of the same malware. The reason behind the five runs is that we wanted to be sure that ScriptGen did not return the right message by chance, and that the experiment can be reliably repeated multiple times. If the extracted model was not sufficient to properly "replay" the network conversation, we added one more network trace to the learning pool and repeated the experiment. The number in the fifth column represents the number of network traces required to create a Finite State Machine that achieved full containment (or its best approximation in the case of Koobface) of the malware sample. The results vary between 9 and 23 traces. These numbers may seem large, but it is important to remember that ScriptGen is completely protocol agnostic and that each experiment was performed starting with an empty protocol model. For example, we discovered that 6 traces is the minimum amount required to properly model a DNS request/response exchange (due to the fact that the response has to contain the same request ID field used in the request).

Table 3 reports the result of our second group of tests. In this second experiment, we focused on incremental learning, i.e., on analyzing current malware in a sandbox environment each time refining our model of the network traffic. We started by executing the samples 3 times, without attempting to replay the traffic. Then we created our first model, and started executing the sample in the sandbox with Mozzie acting as a proxy. Whenever the system was not able to contain the traffic, the requests were forwarded to the real servers, and the FSM updated with the new information. The third column reports the number of time each malware has to be analyzed before the model can achieve full containment for five consequent runs.

Overall, we tested 2 IRC botnets, 1 HTTP botnet, 4 droppers, 1 ransomware, 1 backdoor and 1 keylogger. For these samples, we needed a number of network traces ranging from 4 to 25. The first number seems in contradiction with the lower bound we have previously found. The truth is that this particular sample does not use DNS and thus contact the C&C servers directly by using an hardcoded IP address. For all the other malware that generate DNS traffic the number is definitively higher than the lower bound. On average we need 14 traces to be able to build a good traffic model.

Certainly, large malware analysis systems forced to analyze tens or hundreds of thousand of samples per day cannot afford to repeat the tests 14 times. However, such a high number of new samples collected every day is largely due to the common use of polymorphism and packing techniques by malware writers. Therefore, once a FSM is available for one of the samples in the family, any further variation that preserves the behavior of the program does not require any additional training. Our system could help analyzing polymorphic samples for which the required network infrastructure is not available any more, and that nowadays cannot be tested at all. This, as we already described in the introduction, can improve the result of clustering, and can help malware analysts to properly label those samples that do not work anymore at the time of the analysis. Even better, our system could be used to replicate a specific network scenario that is targeted by a malware infection. Recent years have seen the rise of sophisticated attacks targeting specific environments, such as Stuxnet and Duqu [26, 25]. In these cases, the network traces obtained from the targeted network infrastructure (e.g. traces of interaction in a DCS system in a power generation control system) could be used to build a model of the targeted network environment, allowing the analysis of the malware to be successfully performed inside traditional, and safe, sandboxes.

6. LIMITATIONS

The current prototype of our containment system has several limitations. Some are specific to the way the system has

Sample	Category	Containment	Endpoint Normalization	Traces
W32/Virut	IRC Botnet	FULL	NO	15
PHP/Pbot.AN	IRC Botnet	FULL	NO	12
W32/Koobface.EXT	HTTP Botnet	72%	YES	9
W32/Agent.VCRE	Dropper	FULL	NO	23
W32/Agent.XIMX	Dropper	FULL	YES	10

Table 2: Results of the Offline learning Experiments

Sample	Category	Runs	Containment	Endpoint Normalization
W32/Banload.BFHV	Dropper	23	FULL	NO
W32/Downloader	Dropper	25	FULL	NO
W32/Troj_Generic.AUULE	Ransomware	4	FULL	NO
W32/Obfuscated.X!genr	Backdoor	6	FULL	NO
SCKeylog.ANMB	Keylogger	14	FULL	YES

Table 3: Results of the Incremental learning Experiments

been implemented and some are related to the self-imposed constraints associated to the chosen methodology.

More specifically, we can group the current limitations into three main families:

- The method adopted in this paper is completely protocol-agnostic. While its nature allows us to guarantee our ability to handle custom, undocumented protocols that can be adopted by future malware, it also imposes unnecessary constraints when dealing with simpler and well known protocols such as DNS. We have already seen that our system requires six samples of network interaction to learn how to properly replay a DNS request. The same result could be easily achieved by analyzing only one request, parsing the DNS fields, and extracting the required information in an ad-hoc fashion. However, the goal of this paper was to show how far it is possible to go with a completely generic system. Therefore, our results can be considered as a upper bound, as the system could be easily improved by adding ad-hoc handlers for common and well known protocol interactions.

- Our current prototype is implemented as a network proxy. Even though this approach has some advantages (e.g., it can be easily plugged into any existing sandbox), it makes the analysis of encrypted protocols impossible. However, this is mostly a technical limitation. The same approach could be implemented at the API level, where most of the network traffic is still available in clear text. Most of the malware sandbox environments already hook into the Windows API to extract information about the malware behavior. By adding our system to the hooked network and cryptographic APIs, we could intercept the communication on the host side and achieve full containment also for some encrypted protocols (e.g., the ones based on SSL).

- Our approach is very inefficient when a malware sample exhibits different behaviors independently of the input it receives from the network. For example, if a sample randomly selects the action to perform out of many possible options, Mozzie would require a lot of traces to properly model all possible behaviors. As an extreme case, domain flux techniques (or large pools of domain names like the one described in Section 5) cannot be modeled by our system without requiring protocol-aware heuristics, such as handling the DNS interaction by using a custom DNS service.

7. CONCLUSIONS

This paper addresses the problem of network containment and repeatability in the context of dynamic analysis tools such as sandboxes. As pointed out by previous work [17, 24] malware execution behavior has often a strong dependency with the state and the behavior of external hosts This raises repeatability issues: a malware analysis cannot be reproduced if the state of these external hosts has changed. At the same time, containment concerns are always associated to the communication of malware with external hosts, concerns that have only partially been addressed by the current state of the art.

We have discussed how protocol learning, used in the previous works in the context of service emulation for server-side honeypots, can be successfully used in this new context to provide an emulated, and contained network environment that allows correct execution of malware samples even in presence of undocumented, ad-hoc communication protocols.

We have described the implementation of Mozzie, a network containment system that can be easily adapted to all the existing sandbox environments. According to our experiments, an average of 14 network traces are required by Mozzie to model the traffic by approaching the problem of sandbox network emulation in a completely generic, protocol-agnostic way that can be applied to real-world malware samples.

The benefits of the large-scale application of similar techniques are significant: old malware samples whose C&C infrastructure has been shut down can be analyzed in the same network conditions they were supposed to find when active; in-depth analyses of samples of interest can be carried out in complete isolation, e.g. without direct connectivity to the underlying C&C server and thus without disclosing details on the analysis operation to the bot herders; malware targeting specific network environments (e.g. industrial control systems) can be analyzed in a replica of the network layout they expect to find.

Acknowledgements

This work has been partially supported by the European Commission Seventh Framework Programme (FP7/2007-2013) under grant agreement 257007 and through the FP7-SEC-285477-CRISALIS project.

8. REFERENCES

[1] Scapy. http://www.secdev.org/projects/scapy/, 2003.

[2] nfqueue-bindings. ://www.wzdftpd.net/redmine/projects/nfqueue-bindings/wiki/, 2008.

[3] Anubis. http://anubis.iseclab.org, 2009.

[4] Cwsandbox. http://www.mwanalysis.org, 2009.

[5] Netzob. http://www.netzob.org, 2009.

[6] Cuckoo Sandbox. http://www.cuckoosandbox.org, 2010.

[7] D. Balzarotti, M. Cova, C. Karlberger, C. Kruegel, E. Kirda, and G. Vigna. Efficient Detection of Split Personalities in Malware. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2010.

[8] U. Bayer, C. Kruegel, and E. Kirda. TTAnalyze: A Tool for Analyzing Malware. In *15th European Institute for Computer Antivirus Research (EICAR 2006) Annual Conference*, April 2006.

[9] U. Bayer, P. Milani Comparetti, C. Kruegel, and E. Kirda. Scalable, Behavior-Based Malware Clustering. In *16th Symp. on Network and Distributed System Security (NDSS)*, 2009.

[10] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: automatic extraction of protocol message format using dynamic binary analysis. In *14th ACM conference on Computer and Communications Security*, pages 317–329. ACM New York, NY, USA, 2007.

[11] W. Cui, J. Kannan, and H. J. Wang. Discoverer: Automatic protocol reverse engineering from network traces. In *16th USENIX Security Symposium*, 2007.

[12] W. Cui, V. Paxson, and N. Weaver. GQ: Realizing a system to catch worms in a quarter million places. Technical report, ICSI Tech Report TR-06-004, September 2006.

[13] W. Cui, V. Paxson, N. Weaver, and R. H. Katz. Protocol-independent adaptive replay of application dialog. In *The 13th Annual Network and Distributed System Security Symposium (NDSS)*, February 2006.

[14] D. Inoue, K. Yoshioka, M. Eto, Y. Hoshizawa, and K. Nakao. Malware behavior analysis in isolated miniature network for revealing malware‘s network activity. In *Proceedings of IEEE International Conference on Communications, ICC 2008, Beijing, China, 19-23 May 2008*, pages 1715–1721. IEEE, 2008.

[15] C. Kreibich, N. Weaver, C. Kanich, W. Cui, and V. Paxson. Gq: Practical containment for measuring modern malware systems. In *Proceedings of the ACM Internet Measurement Conference (IMC)*, Berlin, Germany, November 2011.

[16] C. Leita. *SGNET: automated protocol learning for the observation of malicious threats*. PhD thesis, University of Nice-Sophia Antipolis, December 2008.

[17] C. Leita, U. Bayer, and E. Kirda. Exploiting diverse observation perspectives to get insights on the malware landscape. In *DSN 2010, 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2010.

[18] C. Leita and M. Dacier. SGNET: a worldwide deployable framework to support the analysis of malware threat models. In *7th European Dependable Computing Conference (EDCC 2008)*, May 2008.

[19] C. Leita, M. Dacier, and F. Massicotte. Automatic handling of protocol dependencies and reaction to 0-day attacks with ScriptGen based honeypots. In *9th International Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2006.

[20] C. Leita, K. Mermoud, and M. Dacier. Scriptgen: an automated script generation tool for honeyd. In *21st Annual Computer Security Applications Conference*, December 2005.

[21] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution. In *15th Annual Network and Distributed System Security Symposium*, San Diego, CA, February 2008.

[22] Z. Lin and X. Zhang. Deriving input syntactic structure from execution. In *16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Atlanta, GA, USA, November 2008.

[23] S. Needleman and C. Wunsch. *A general method applicable to the search for similarities in the amino acid sequence of two proteins*. J Mol Biol. 48(3):443-53, 1970.

[24] C. Rossow, C. J. Dietrich, H. Bos, L. Cavallaro, M. van Steen, F. C. Freiling, and N. Pohlmann. Sandnet: Network Traffic Analysis of Malicious Software. In *1st Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, April 2011.

[25] Symantec. The Stuxnet worm. http://go.symantec.com/stuxnet.

[26] Symantec. W32.Duqu, the precursor to the next Stuxnet. http://go.symantec.com/duqu.

[27] Symantec. W32.Koobface. http://www.symantec.com/security_response/writeup.jsp?docid=2008-080315-0217-99.

[28] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, fidelity, and containment in the Potemkin virtual honeyfarm. *ACM SIGOPS Operating Systems Review*, 39(5):148–162, 2005.

[29] G. Wondracek, P. M. Comparetti, C. Kruegel, and E. Kirda. Automatic network protocol analysis. In *15th Annual Network and Distributed System Security Symposium (NDSS'08)*, 2008.

[30] K. Yoshioka, T. Kasama, and T. Matsumoto. Sandbox analysis with controlled internet connection for observing temporal changes of malware behavior. In *2009 Joint Workshop on Information Security (JWIS 2009)*, 2009.

Lines of Malicious Code: Insights Into the Malicious Software Industry

Martina Lindorfer
Secure Systems Lab, Vienna
University of Technology
mlindorfer@seclab.tuwien.ac.at

Alessandro Di Federico
NECSTLab, PoliMi, Italy
alessandro.difederico@mail.polimi.it

Federico Maggi
NECSTLab, PoliMi, Italy
fmaggi@elet.polimi.it

Paolo Milani Comparetti
Lastline, Inc.
pmilani@lastline.com

Stefano Zanero
NECSTLab, PoliMi, Italy
zanero@elet.polimi.it

ABSTRACT

Malicious software installed on infected computers is a fundamental component of online crime. Malware development thus plays an essential role in the underground economy of cyber-crime. Malware authors regularly update their software to defeat defenses or to support new or improved criminal business models. A large body of research has focused on detecting malware, defending against it and identifying its functionality. In addition to these goals, however, the analysis of malware can provide a glimpse into the software development industry that develops malicious code.

In this work, we present techniques to observe the evolution of a malware family over time. First, we develop techniques to compare versions of malicious code and quantify their differences. Furthermore, we use behavior observed from dynamic analysis to assign semantics to binary code and to identify functional components within a malware binary. By combining these techniques, we are able to monitor the evolution of a malware's functional components. We implement these techniques in a system we call BEAGLE, and apply it to the observation of 16 malware strains over several months. The results of these experiments provide insight into the effort involved in updating malware code, and show that BEAGLE can identify changes to individual malware components.

Categories and Subject Descriptors

K.6.5 [**Security and Protection**]: Invasive software

General Terms

Security

Keywords

Malware; Evolution; Downloaders; Similarity.

1. INTRODUCTION

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

In parallel with the development of cybercrime into a large underground economy driven by financial gain, malicious software has changed deeply. Originally, malicious software was mostly simple self-propagating code crafted primarily in low-level languages and with limited code reuse. Today, malicious software has turned into an industry that provides the tools that cybercriminals use to run their business [30]. Like legitimate software, malware is equipped with auto-update functionality that allows malware operators to deploy arbitrary code to the infected hosts. New malware versions are frequently developed and deployed; Panda Labs observed 73,000 new malware samples per day in 2011 [5]. Clearly, the majority of these samples are not really new software but rather repacks or incremental updates of previous malware. Malware authors update their code in an endless arms race against security countermeasures such as anti-virus engines and SPAM filters. Furthermore, to succeed in a crowded, competitive market they "innovate", refining the malware to better support cybercriminals' modus operandi or to find new ways to profit at the expense of their victims.

Understanding how malware is updated over time by its authors is thus an interesting and challenging research problem with practical applications. Previous work has focused on constructing the *phylogeny* of malware [15, 20]. Instead, we would like to observe subsequent versions of a malware and automatically characterize its evolution. As a first step, quantifying the differences between versions can provide an indication of the development effort behind this industry, over the observation period.

To provide deeper insight into malicious software and its development, we need to go a step further and identify how the changes between malware versions relate to the functionality of the malware. This is the main challenge that this work addresses. We propose techniques that combine dynamic and static code analysis to identify the component of a malware binary that is responsible for each behavior observed in a malware execution, and to measure the evolution of each component across malware versions.

We selected 16 malware samples from 11 families that included auto-update functionality, and repeatedly ran them in an instrumented sandbox over a period of several months, allowing them to update themselves or download additional components. As a result of dynamic analysis, we obtain the unpacked malware code and a log of its system-level activity. We then compare subsequent malware versions to identify code that is shared with previous versions, and code that was added or removed. From the system-level activity we infer high-level behavior such as "downloading and executing a binary" or "harvesting email addresses". Then, we identify the binary code that implements this observed functionality, and track how it changes over time. As a result, we are able to observe not only the

overall evolution of a malware sample, but also the evolution of its individual functional components.

In summary, this paper makes the following contributions:

- We propose techniques for binary code comparison that are effective on malware and can also be used to contrast a single binary against multiple others, such as against all previous versions of a malware or against a dataset of benign code.
- We propose techniques that use behavior observed from dynamic analysis to assign semantics to binary code. With these techniques, we can identify functional components in a malware sample and track their evolution across malware versions.
- We implement these techniques in a tool called BEAGLE, and use it to automatically monitor 16 malware samples from 11 families over several months and track their evolution in terms of code and expressed behavior. Our results, based on over one thousand executions of 381 distinct malware instances, provide insight into how malware evolves over time and demonstrate BEAGLE's ability to identify changes to malware components.

2. EVOLUTION OF MALICIOUS CODE

Malware is the underlying platform for online crime: From SPAM to identity theft, denial of service attacks or fake-antivirus scams [4], malicious software surreptitiously installed on a victim's computer plays an essential role in criminals' online operations. One key reason for the enduring success of malicious software is the way it has adapted to remain one step ahead of defenses.

Cybercriminals are under constant pressure, both from the security industry, and—as in any market with low barriers to entry—from competing with other criminals. As a result, malicious software is constantly updated to adapt to this changing environment.

The most obvious form of adversarial pressure that the security industry puts on malware authors comes from antivirus (AV) engines: Being detected by widely-deployed AVs greatly reduces the pool of potential victims of a malware. Thus, malware authors strive to defeat AV detection by using packers [29]—also known as "crypters". AV companies respond by detecting the unpacking code of known crypters. The result is that a part of the malicious software industry has specialized in developing crypters that are not yet detected by AVs: Interestingly, Russian underground forums advertise job openings for crypter developers with monthly paychecks of 2,000 to 5,000 US Dollars [21].

This is however only one aspect of the ongoing arms race. SPAM bots try to defeat SPAM filters using sophisticated template-based SPAM engines [34]. Meanwhile, botnets' command and control (C&C) infrastructure is threatened by take downs, so malware authors experiment with different strategies for reliably communicating with their bots [35]. Similarly, the security measures deployed at banking websites and other online services, such as two-factor authentication, require additional malicious development effort. For this, malware authors embed code into the victim's browser that is targeted at a specific website, for instance to mislead him into sending money to a different account than the one he intended to. A plugin implementing such a "web inject" against a specific website for a popular bot toolkit can be worth 2,000 US Dollars [21].

The need to remain one step ahead of defenses is only one reason for malware's constant evolution. Cybercriminals strive to increase their profits—which are threatened by trends such as the declining prices of stolen credentials [22]—by developing new business models. These often require new or improved malware features. As an example, one sample in our dataset implements functionality to simulate a system malfunction on the infected host—presumably to try to sell the victim fake AV or utility software that will "solve" the problem, and another one is able to steal from Bitcoin wallets.

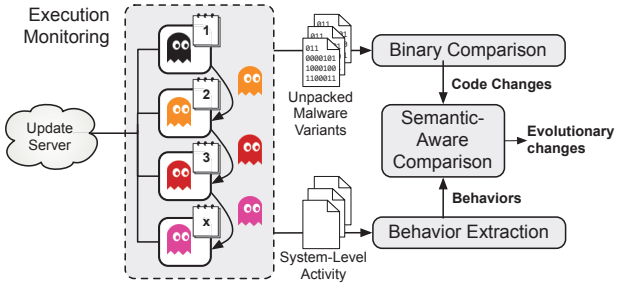


Figure 1: Overview of BEAGLE.

From an economic perspective, we would like to know how expensive it is for a malware author to develop a new feature, as well as how much effort he needs to invest into defeating a security countermeasure. In the absence of direct intelligence on a malware's developers practices, the malware code itself is the richest available source of information on the malware development process. Human analysts routinely analyze malware binaries to understand not only what they can do but also to get a glimpse into the underground economy of cybercrime. In this work, we develop techniques to automate one aspect of this analysis: Measuring how malware changes over time and how these changes relate to the functionality it implements.

3. APPROACH

We implement the approach described in this section in a system we called BEAGLE[1]. Fig. 1 shows an overview of our approach.

BEAGLE first dynamically analyzes a malware sample by running it in an instrumented sandbox, as detailed in §4.1. We allow the sample to connect to its C&C infrastructure to obtain an updated binary. This way, we capture new versions of the analyzed malware. Later, we analyze these updated samples by also executing them in our analysis sandbox. In addition to recording the sample's system- and network-level behavior, the sandbox also acts as a simple generic unpacker [32], and records the unpacked version of the malware binary and of any other executables that are run on the system. This includes malicious code that is injected into benign processes. This allows us to analyze the deobfuscated binary code and compare it across versions with static code-analysis techniques.

Then, BEAGLE compares subsequent versions of a malware's code to quantify the overall evolution of the malware code. Our approach to binary code comparison is based on the control-flow graph's (CFG) structural features, as described in §4.2. This first comparison provides us with an overall measure of how much the code has changed and how much new code has been added, but it offers no insight into the *meaning* of the observed changes.

To attach semantics to the observed changes, we take advantage of the behavioral information provided by our analysis sandbox. The sandbox provides us with a trace of low-level operations performed by the malware. In the behavior extraction phase, discussed in §4.3, we analyze this system-level activity to detect higher-level behaviors such as "sending SPAM", "downloading and executing a binary from the Internet". These behaviors are detected based on a set of high-level rules. In addition to these behaviors that have well-defined semantics (as determined by the human analyst who wrote the detection rule), we also detect behaviors with a-priori unknown semantics, by grouping related system-level events into

[1]HMS Beagle was the ship that Charles Darwin sailed on in the voyage that would provide the opportunity for many of the observations leading to his development of the theory of evolution.

unlabeled behaviors. Once a behavior has been detected, we employ a lightweight technique to map the observed behavior to the code that is responsible for it, and tag individual functions with the set of behaviors they are involved in.

Last, we perform semantic-aware binary comparison as described in §4.4. For this, we combine the results of the behavior-detection step with our binary-comparison techniques to analyze how the code's evolution relates to the observed behaviors. Thus, we are able to monitor when each behavior appears in a malware's codebase and to quantify the frequency and extent of changes to its implementation. The goal is to provide insight into the semantics of the code changes and help the analyst to answer questions such as "What is the overall amount of development effort behind a malware family?", "Within a family, which components are most actively developed?" or "How frequently is a specific component tweaked?".

4. SYSTEM DESCRIPTION

BEAGLE has four modules. The Execution Monitoring module is a sandbox that logs the relevant actions that each sample performs while running (e.g., call stack, system calls, sockets opened, system registry modifications, file creation). The Behavior Extraction module analyzes these logs for each malware sample and extracts high-level behaviors using a set of rules and heuristics. In parallel, the Binary Comparison module disassembles the unpacked binary code. Then, it analyzes the code of each malware sample to find added, removed and shared portions of the CFG across samples, and "labels" these portions with the relative high-level behavior extracted at runtime. The Semantic-Aware Comparison module monitors how the labeled code evolves over time.

4.1 Execution Monitoring

The first step is to obtain subsequent variants of that malware family. One approach would be to rely on the labeling of malware samples by AV engines, and select samples from a specific family as they become available over time. However, this approach has two problems. First, AV engines are not very good at classifying malware largely because their focus is on detection rather than classification [6,25]. Second, different samples of a malware family may be independent forks of a common code base, which are developed and operated by different actors.

The approach we use in BEAGLE is to take advantage of the auto-update functionality included in most modern malware. Specifically, by running malware in a controlled environment and letting it update itself, we can be confident that the intermediate samples that we obtain represent the evolution over time of a malware family—as deployed by a specific botmaster.

Stateful Sandbox. A malware analysis sandbox typically runs each binary it analyzes in a "clean" environment: Before each execution, the state of the system in the sandbox is reset to a snapshot. This ensures that the sample's behavior and other analysis artifacts are not affected by previously-analyzed samples. To allow samples to install and update themselves, however, we need a different approach. A simple yet extremely inefficient solution would be to let the malware run in the sandbox for the entire duration of our experiments. Running a malware sample for months also increases the risk that, despite containment measures, it may cause harm. Instead, we rely on malware's ability to persist on an infected system. Each sample is allowed to run for only a few minutes. At the end of each analysis run, BEAGLE captures the state of the system, in the form of a patch that contains file-system and registry changes to the "clean" system state. Additionally, we detect a malware sample's persistence mechanism by monitoring known auto-start locations [36]. When we want to re-analyze the sample, we start by applying the patch to the "clean" snapshot. Then, we trigger the detected persistence mechanism, so that we effectively simulate a reboot, thus continuing the analysis with updated versions. Hence, in the first execution we observe the installation and update functionality of the original malware sample. In the following executions we observe the updated variants and additional updates.

Generic Unpacking. BEAGLE's analysis sandbox also captures the deobfuscated binary code by acting as a simple, generic unpacker. In its current implementation, BEAGLE simply captures and dumps all code found in memory at process exit or at the end of the analysis run. We do not restrict this to the initial malware process, but also include all code from binaries started by the malware or of processes in which the malware has injected code. For instance, one of the ZeuS samples in our dataset downloads code from a C&C server and injects it into a the `explorer.exe` process. In such a case, BEAGLE also dumps an image of the `explorer.exe` process that includes the injected code.

This simple approach to unpacking, although sufficient for the purpose of our experiments, could be defeated by more advanced packing approaches where unpacked code is re-packed or deleted immediately after use. To address this limitation, BEAGLE could be extended to incorporate more advanced generic unpacking techniques [26, 32].

Sandbox Implementation. BEAGLE's sandbox is an extension of Anubis [8], a dynamic malware analysis sandbox based on the whole-system emulator Qemu. Anubis captures a malware's behavior in the sandbox at the API level, producing a log of the invoked system and API calls, including parameters. Furthermore, Anubis uses dynamic taint tracing to capture data flow dependencies between these calls [9]. In order to facilitate attribution of behavior observed in the sandbox to the code responsible for it, we extended the sandbox to log the call stack corresponding to each API call.

4.2 Binary Comparison

The next component of BEAGLE compares binaries and identifies the code that is shared between versions, the code that was added, or removed. This allows us to quantify the evolution of malicious code by measuring the size of code changes and computing a similarity score between malware variants. More importantly, as discussed in §4.4, we combine this information with code semantics inferred from dynamic behavior to gain an understanding of how evolution relates to malware functionality.

Code Fingerprints. Our technique for binary comparison relies on the structure of the intra-procedural CFG, extending work from [23]. The CFG is a directed graph where nodes are basic blocks and an edge between two nodes indicates a possible control flow transfer (e.g., a conditional jump) between those basic blocks. Following [23], nodes in the CFG are colored based on the classes of instructions that are present in each node, and the problem of finding shared code between two binaries is reduced to searching for isomorphic k-node subgraphs (we use $k = 10$ following [23]). As this problem is intractable, [23] proposed an efficient approximation that relies on extracting a subset of a CFG's k-node connected subgraphs and normalizing them. Each normalized subgraph is a concise representation of a code fragment. In practice, the normalized subgraph is hashed to generate a succinct fingerprint. By matching fingerprints generated from different code samples, we are able to efficiently detect similar code.

As shown by Kruegel et al. [23], these fingerprints are to some extent resistant to code metamorphism, and are effective for detecting code reuse in malware binaries [28]. Here, we take this a step forward and use them to locate the shared code between successive

malware variants. For this, given an unpacked malware binary M, we disassemble it, extract the colored CFG and compute the corresponding set of CFG fingerprints $\mathcal{F}_M$. For each fingerprint f, we also keep track of the addresses $b_M(f)$ of the set of basic blocks it was generated from. For a set of fingerprints $\mathcal{F}$, we indicate the corresponding basic blocks in sample M with $\mathcal{B}_M(\mathcal{F}) = \bigcup_{f\in\mathcal{F}} b_M(f)$. To compare two samples M and N, we compute the intersection of the corresponding fingerprints $I_{M,N} = \mathcal{F}_M \cap \mathcal{F}_N$. The basic blocks corresponding to the fingerprints in this set in either sample correspond to code that is common to M and N. We call them *shared* basic blocks. In sample M, the set of basic blocks that is shared with N is thus $\mathcal{S}_M(M,N) = \mathcal{B}_M(I_{M,N})$. If M and N are two successive variants of a malware $\mathcal{A}_N(M,N) = \mathcal{B}_N \setminus \mathcal{S}_N(M,N)$ (short, $\mathcal{A}(M,N)$) is the set of basic blocks *added* in the new sample N, whereas $\mathcal{R}_M(M,N) = \mathcal{B}_M \setminus \mathcal{S}_M(M,N)$ (short, $\mathcal{R}(M,N)$) is the set of basic blocks *removed* from the old sample M.

Code Whitelisting. The unpacked code produced by our sandbox may include code unrelated to the malware. One reason is that malware may inject code into legitimate processes, that will then be included into our analysis. Furthermore, a malware process sometimes loads system dynamically linked libraries (DLLs) directly into its address space, without using the standard API functions for loading DLLs. In both cases, the unpacked binaries produced by our analysis sandbox will include code that is not part of the malware.

To exclude this code from analysis, we rely on a whitelisting approach. For this, we identify all code (executables and DLLs) in the "clean" sandbox, and compute the set $\mathcal{W}$ of all CFG fingerprints found in this code. In our analysis of each malware sample M, we then identify the basic blocks $\mathcal{B}(\mathcal{W})$ that match these fingerprints. We call them *whitelisted* basic blocks, and do not take them into account for further analysis. An additional benefit of whitelisting code from a clean system is that this can also eliminate code from standard system libraries that has been statically linked into a malware binary.

Similarity and Evolution. We compute the similarity between two malware samples with a variant of the Jaccard set similarity:

$$J(M,N) := \frac{|\mathcal{S}^\star(M,N)|}{|\mathcal{S}^\star(M,N)| + |\mathcal{A}(M,N)| + |\mathcal{R}(M,N)|} \qquad (1)$$

This is roughly the number of shared basic blocks over the total of shared, added and removed basic blocks. The number of shared basic blocks in the two samples $|\mathcal{S}_N(M,N)|$ and $|\mathcal{S}_M(M,N)|$ may differ slightly, because multiple identical k-node subgraphs may exist in a single sample. We mitigate this by picking $|\mathcal{S}^\star(M,N)|$, which is the maximum between the two.

In addition to comparing pairs of samples, we would like to contrast a new malware sample against all previously observed variants, and identify the code that is new in the latest variant. Measuring this code provides the most direct measure of the malware authors' development effort for this variant. For this, we compute the set $\mathcal{F}_{M_1,..,M_{t-1}}$ of fingerprints found in the first $t-1$ samples, and identify the *new basic blocks* $\mathcal{N}_{M_t} = \mathcal{B}_{M_t} \setminus \mathcal{B}_{M_t}(\mathcal{F}_{M_1,..,M_{t-1}})$.

4.3 Behavior Extraction

Automatically understanding the purpose and semantics of binary code is a challenging task. The system-level behavior of a malware sample in an analysis sandbox, however, can be more readily interpreted. To detect specific patterns of malicious behavior previous work [16, 27] has started from system-level events, enriched with data flow information. Martignoni et al. in [27] frame this as a "semantic gap" problem, and propose a technique to detect high-level behavior from system-level events using a hierarchy of manually crafted rules. In the absence of prior knowledge of a pattern of malicious behavior, on the other hand, no such rules are available. As an alternative to leveraging prior knowledge, researchers have used data flow to link individual system-level events into graphs, and applied data mining techniques to a corpus of such graphs to learn to detect malware [11] or to identify its C&C communication [17].

We aim to make our observation of the evolution of malware functionality as complete and insightful as possible. Therefore, we assign semantics to observed behavior that matches known patterns, but also take into account behaviors for which no high-level meaning can be automatically established. In this work, we call the former *labeled*, and the latter *unlabeled* behavior.

Unlabeled Behavior. An unlabeled behavior is a connected graph of system-level events observed during the execution of a sample. Nodes in this graph represent system or API calls, whereas the edges represent data flow dependencies between them. As data flow dependencies are lost when files are written to disk, we also connect nodes that operate on the same file system resources.

Our purpose in taking unlabeled behavior into account is to identify components of the malware and measure their evolution, even though we do not (yet) know what functionality they implement.

Labeled Behavior. A labeled behavior consists of one or more unlabeled behaviors—or, in other words, connected subgraphs of system-level events—such that they match a manually-crafted specification of a known malware behavior. These specifications can take into account the API calls involved, their arguments, as well as the data flow between them. For instance, we define the `DOWNLOAD_EXECUTE` behavior as any data flow dependency from the network to a file that is later executed; another example is the `AUTO_START` behavior, which we define as a write to any of a set of known autostart locations.

Behaviors that remain unlabeled can be examined by an analyst who can assign semantics to them and define behavior specifications for detecting them. Thus, BEAGLE's knowledge-base of behavior specifications grows over time to cover a broader spectrum of malware functionality. As we show in §5.6, a relatively small number of manually-written behavior specifications was sufficient to cover a significant fraction of the malware code that was executed during our experiments.

4.4 Semantic-Aware Comparison

The goal of this step is to attach meaning to the overall changes in the binary code. Essentially, we divide the malware program into a number of functional components. To this end, BEAGLE starts from the behaviors observed at runtime, as discussed in §4.3, and identifies the binary code that is responsible for each observed behavior. By tracking the evolution of this code across malware versions, we are able to measure the evolution of malware's functional components.

Mapping Behavior to Code. The output of the behavior extraction phase is, for each behavior observed in a sample's execution, a sequence of system or API calls with the corresponding call stack. BEAGLE next tries to identify the code responsible for this behavior. BEAGLE works at function granularity: It identifies the set of functions in the unpacked binary that are involved in that behavior, and "tags" them with the behavior. A single function may be tagged with multiple behaviors (e.g., a utility function that is re-used across different functional components). To identify the functions involved in a behavior, we use static analysis to identify a code path that could have been responsible for the observed sequence of calls (taking into account the corresponding call stacks). To do so, we resolve the path between any two consecutive calls by recursively looking up code references to the target function until we find the source function.

Family Name and Label	Source	1st day	Days	Executions	MD5s	Lifespan
Banload TrojanDownloader:Win32/Banload.ADE	(1)	2012-01-31	87	78	3	2.00/83.00/29.33/37.95
Cycbot Backdoor:Win32/Cycbot.G	(1)	2011-09-15	73	73	69	1.00/73.00/2.04/8.60
Dapato Worm:Win32/Cridex.B	(2)	2012-02-24	65	62	25	1.00/43.00/4.60/8.31
Gamarue Worm:Win32/Gamarue.B	(2)	2012-02-10	78	77	19	1.00/76.00/8.47/16.44
GenericDownloader TrojanDownloader:Win32/Banload.AHC	(1)	2012-01-31	82	79	5	2.00/69.00/16.80/26.16
GenericTrojan Worm:Win32/Vobfus.gen!S	(1)	2012-02-07	76	73	55	1.00/44.00/2.71/6.32
Graftor TrojanDownloader:Win32/Grobim.C	(1)	2012-02-17	37	39	22	1.00/17.00/6.00/5.53
Kelihos TrojanDownloader:Win32/Waledac.C	(2)	2012-03-03	56	38	8	1.00/54.00/21.00/22.88
Llac Worm:Win32/Vobfus.gen!N	(1)	2012-02-07	32	33	82	1.00/10.00/1.49/1.71
OnlineGames Worm:Win32/Taterf.D	(1)	2011-09-02	87	80	47	1.00/38.00/3.94/7.28
ZeuS PWS:Win32/Zbot.gen!AF `1be8884c7210e94fe43edb7edebaf15f`	(3)	2012-02-09	79	78	6	1.00/78.00/26.67/28.70
ZeuS PWS:Win32/Zbot `9926d2c0c44cf0a54b5312638c28dd37`	(3)	2012-02-15	74	73	4	1.00/50.00/18.50/19.63
ZeuS PWS:Win32/Zbot.gen!AF* `c9667edbbcf2c1d23a710bb097cddbcc`	(3)	2012-02-23	66	63	6	1.00/36.00/11.00/13.43
ZeuS PWS:Win32/Zbot.gen!AF* `dbedfd28de176cbd95e1cacdc1287ea8`	(3)	2012-02-09	79	78	4	1.00/78.00/20.25/33.34
ZeuS PWS:Win32/Zbot.gen!AF* `e77797372fbe92aa727cca5df414fc27`	(3)	2012-02-10	79	77	5	1.00/77.00/16.20/30.40
ZeuS PWS:Win32/Zbot.gen!AF* `f579baf33f1c5a09db5b7e3244f3d96f`	(3)	2012-03-03	57	55	11	1.00/30.00/5.64/9.75

Table 1: Dataset. The labels in the first columns are based on Microsoft AV naming convention. The MD5 column is the number of distinct binaries encountered. Lifespan is the duration in days of the interval in which an MD5 was observed (min/max/mean/stdev).

We use the addresses in the call stack as landmarks this path should traverse and in case the dynamic path cannot be resolved statically. We tag all functions in the identified path as part of the behavior.

Working at function granularity is a design decision that trades off some precision in delimiting functional components, to achieve performance compatible with a large-scale experiment. As discussed in §4.1, our modified sandbox logs events at the system API level. Previous work that performed a similar mapping of behavior to code at instruction granularity [28], on the other hand, relied on a sandbox logging each executed basic block.

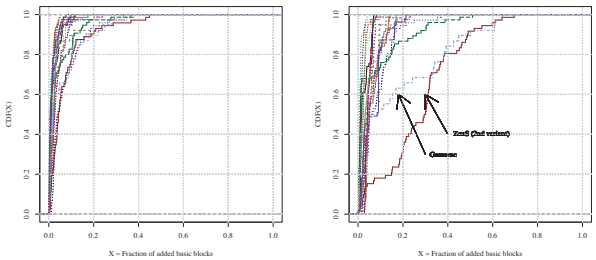
Behavior Evolution. The set of functions that implement a behavior is a functional component of a malware instance. By comparing the components that implement a behavior in successive versions of a malware, we can observe the evolution of that functionality over time. This allows us to get an idea of the development effort involved in updating this functionality by measuring the amount of new code, as well as quickly identifying significant updates to the malicious functionality that may warrant further inspection. For this, we apply the techniques discussed in §4.2 to successive versions of a component, instead of considering entire unpacked binaries. Among the versions of a component observed throughout our experiments, we select the largest implementation by number of basic blocks and call it the *reference behavior*.

Dormant Functionality. Like any dynamic code analysis approach, a limitation of BEAGLE is incomplete code coverage. In a typical execution, a malware sample will reveal only a fraction of the functionality it is capable of: For instance, a bot will send SPAM only if instructed to do so by the botnet's C&C infrastructure. Thus, the techniques described above will identify the presence of each functional component only in some of a sample's executions, even though the code implementing the functionality is present throughout our experiments. This limits our visibility in the component's evolution. In the limit, if a behavior is observed only once, we do not see any evolution. To be able to track evolution in a more complete way, we use the CFG fingerprints from §4.2 to identify a component even in executions where it is *dormant* and the corresponding behavior cannot be observed. For this, we identify the dormant components by locating the functions in a sample that match fingerprints from an *active* (non-dormant) component in another execution.

5. EXPERIMENTAL EVALUATION

5.1 Setup

We run BEAGLE on a desktop-class, dual-core machine with 4GB of RAM, and execute each sample for 15 minutes approximately



(a) $t-1$ vs. t (b) t_0 vs. t

Figure 3: CDF of added basic blocks per family. Day-to-day changes (a) are concentrated around low values for all the families, whereas in the long run (b) each family evolves distinctively, showing different development efforts.

once a day, depending on the workload of the sandbox. Each analysis continues from the state of the previous day's execution in order to analyze only the updated versions. Since we want to observe malware updating itself, we cannot run it in a completely isolated environment, but need to allow it to access the C&C infrastructure from which to obtain updates. To prevent malware from causing harm, we employ containment measures such as redirecting "dangerous" protocols to a local honeypot and limiting bandwidth and connections. These measures cannot guarantee that the malware we run will never cause harm (0-day attacks are especially hard to recognize and block), but we believe that they are sufficient in practice if combined with a prompt response to any abuse complaints (we did not receive any complaints during the course of our experiments).

5.2 Dataset

We selected samples from three different sources: (1) Recent submissions to Anubis for which the data flow detection of Jackstraws [17] indicated download & execute behavior. (2) Malware variants from the top threats according to the Microsoft Malware Protection Center [2] (3) ZeuS samples from ZeuS Tracker [3]. We then discarded samples that showed no update activity in our environment.

As summarized in Tab. 4.3, we analyzed the evolution of 16 samples from 11 families between September 2011 and April 2012. We stopped the analysis and discarded a sample after it failed to contact its C&C server for more than two weeks. Overall, we analyzed a total of 1,023 executions of 381 distinct malware binaries.

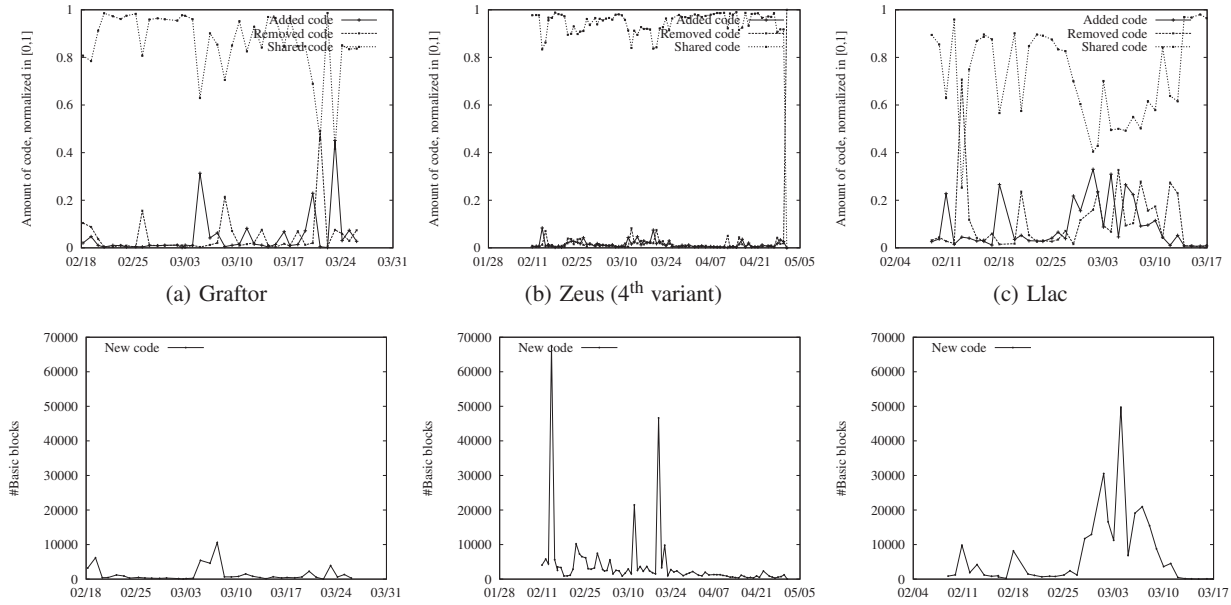


Figure 2: Added, removed and shared code over time. The first row shows the day-to-day changes in the code (i.e., we compare subsequent pairs of samples in the same family), whereas the second row shows the new code. Other families such as Gamarue and GenericDownloader, omitted for space limits, also exhibit interesting evolutions.

5.3 Validation

BEAGLE automatically finds differences between hundreds of malware samples in a few hours on a desktop-class computer. Clearly, an in-depth understanding of the code differences would still require a reverse engineer, which BEAGLE cannot possibly substitute. However, BEAGLE provides valuable guidance to quickly decide what are the interesting changes between malware releases to focus manual inspection. That said, it is hard to assess the correctness of BEAGLE, mainly because we do not have the source code of successive malware versions, and lack a ground truth on the semantics of malware components and their comparison.

The binary comparison component of BEAGLE described in §4.2, however, can be validated by comparing its results against established tools for binary code comparison. For this, we use BinDiff [1, 13], the leading commercial tool for binary comparison and patch analysis, and compare the similarity scores it produces when comparing pairs of samples to those from BEAGLE. Note that BinDiff is based on a completely different approach and uses a number of proprietary heuristics for comparison, and relies on program's call graph, which is not taken into account by our tool. Although the absolute value of BinDiff's similarity score and BEAGLE's similarity score differ, we were able to find a linear transformation[2] from one value to another with a low residual mean square error (6.3% on average, with a peak of 17% for `GenericTrojan`). Note however that BinDiff cannot efficiently be used to contrast a binary against multiple others. BEAGLE's binary comparison component, on the other hand, can be used to contrast a sample against all previous versions (to detect new code), or against a library of benign software (for whitelisting), as discussed in §4.2.

5.4 Overall Changes

In Fig. 2 we show the timelines resulting from comparing samples within three families: Graftor, ZeuS (4th variant), and Llac. We only show three timelines because of space limitations. For the top row of the figure, we consider each consecutive pair of samples of a malware, that is at time t and $t-1$. Then, as described in §4.2, we calculate the amount of added, removed and shared code (Eq. (1)), expressed in distinct basic blocks, between the two samples, and normalized to the total number of distinct basic blocks. For the bottom row, we calculate the absolute amount of code that was never found in any of the previous samples.

These timelines show that overall, day-to-day changes in malware code are relatively small: As would be expected, most new malware

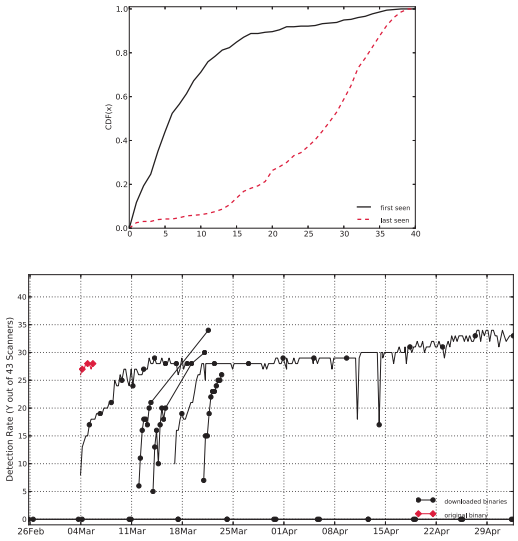


Figure 4: (a) CDF of the detection rate of malware samples among 43 AV engines, at the time each sample is observed for the first and last time in our experiments. (b) Detection rate of Kelihos binaries over time—one line per distinct binary.

[2]To this end, we used R's linear model fitting functions (`lm()` and `poly()`).

FAMILY NAME	%TAGGED	%LABELED	%RATIO	%ADDED	%REMOVED	%SHARED	NEW	#LABELS
Banload	7.31 ± 1.70	6.68 ± 0.75	91.43	2.48 ± 2.96	2.83 ± 3.10	94.69 ± 3.75	176.2 ± 409.2	5
Cycbot	32.36 ± 2.40	31.23 ± 2.95	96.50	10.59 ± 10.36	10.30 ± 10.42	79.11 ± 12.80	1361.4 ± 3937.2	11
Dapato	2.81 ± 1.22	1.15 ± 0.55	40.90	5.15 ± 5.14	5.57 ± 5.63	89.28 ± 7.48	2402.9 ± 7165.3	4
Gamarue	15.90 ± 14.06	14.06 ± 13.40	88.42	12.08 ± 8.16	12.50 ± 9.32	75.41 ± 11.57	2500.1 ± 7747.2	12
GenericDownloader	9.10 ± 1.93	8.58 ± 1.59	94.30	9.80 ± 9.85	9.58 ± 8.81	80.62 ± 12.48	3330.6 ± 7367.8	6
GenericTrojan	22.94 ± 11.05	20.18 ± 10.69	87.97	16.66 ± 16.15	17.03 ± 15.15	66.31 ± 18.76	4974.1 ± 14339.6	11
Graftor	12.66 ± 6.20	9.58 ± 4.70	75.70	6.47 ± 10.40	6.84 ± 9.96	86.69 ± 13.48	682.0 ± 1662.8	4
Kelihos	24.20 ± 2.24	24.09 ± 2.26	99.53	5.18 ± 8.69	5.60 ± 10.10	89.23 ± 12.64	2145.3 ± 4065.3	12
Llac	19.13 ± 14.25	19.11 ± 14.26	99.91	12.82 ± 12.53	14.45 ± 14.70	72.73 ± 19.05	3323.3 ± 7899.1	10
OnlineGames	2.18 ± 0.30	1.96 ± 0.21	89.97	3.35 ± 3.12	3.37 ± 3.12	93.28 ± 5.44	420.0 ± 718.0	9
Zeus	8.37 ± 2.59	6.15 ± 1.32	73.44	2.10 ± 2.24	3.59 ± 11.27	94.31 ± 11.28	1910.8 ± 6148.0	11
Zeus	8.26 ± 1.56	6.44 ± 1.14	78.00	3.65 ± 3.07	5.25 ± 11.85	91.09 ± 12.41	4086.0 ± 11936.3	12
Zeus	10.45 ± 2.67	7.91 ± 2.49	75.73	2.61 ± 2.20	4.51 ± 12.64	92.88 ± 12.47	2234.5 ± 7117.9	11
Zeus	8.55 ± 2.15	6.53 ± 1.19	76.41	2.55 ± 2.51	3.93 ± 11.26	93.52 ± 11.35	2013.6 ± 6874.5	12
Zeus	8.82 ± 1.79	7.73 ± 1.36	87.65	3.12 ± 2.78	4.57 ± 11.33	92.32 ± 11.46	3245.9 ± 7456.3	12
Zeus	7.44 ± 1.31	6.41 ± 0.88	86.06	2.24 ± 2.51	4.53 ± 13.46	93.23 ± 13.46	2523.9 ± 6834.9	13

Table 2: Overall tagged and labeled code (in each version), added, removed, shared code (between consecutive versions), and new code (with respect to all previous versions) for each family (mean±variance, measured in basic blocks). #Labels is the number of distinct behavior labels detected throughout the versions.

versions are incremental updates that reuse most of the code. New code is largely concentrated in a smaller number of peaks, that indicate significant updates to the malware code base. For some families, the amount of brand new code in some of these peaks is significant, up to for instance 50,000 new basic blocks for `Llac`.

Fig. 3 shows a CDF of the similarity of day-to-day differences between successive malware versions (3(a)) and between each version and the first analyzed version (3(b)). Comparing the two graphs clearly shows that while daily updates mostly consist of small changes, for some families the cumulative effect of these small changes eventually result in binaries that are very different from the original sample. This long term evolution varies a lot across the families in our dataset. In Fig.3(b) we highlight `Zeus` (2nd variant) and `Gamarue` that show particularly large cumulative changes.

Tab. 2 summarizes our results, which confirm that, in the majority of cases, the malware writers reuse a significant amount of code when they release day-to-day updates. Remarkably, for a few families the new code added in day-to-day changes accounts for around 10% of the entire malware code on *average*.

5.5 AV Detection

In §2 we suggested that AV engines are one of the main reasons malware authors frequently update their code. During the course of our experiments, we scanned all observed binaries with 43 Anti-Virus engines by using the VirusTotal service. Fig. 4a shows that, as expected, the detection rate of AV engines on a set of binaries—in this case, all binaries executed in our experiments—is generally lower at the beginning of their life cycle than towards the end. More interestingly, we found that in some families, such as Kelihos, as shown in Fig. 4b, the malware writers seem to release a new binary as soon as previous binaries get detected by AV engines. Indeed, whenever a larger number of AV engines start detecting a sample, the malware writers unleash a new, unknown binary, causing a sudden drop in the detection rate.

5.6 Code Behavior

As described in §4.3, we specify behavior as graphs of API calls, connected by data flow, and can take into account API call parameters. Depending on the granularity at which BEAGLE should track changes, an analysts can label behavior by rules that comprise only one function such as `RegSetValue(HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run,*)` for (one variant of) the `AUTO_START` behavior, or more complex rules such as `InternetOpenUrl|connect -> InternetReadFile|recv -> WriteFile -> WinExec|CreateProcess` for the `DOWNLOAD_EXECUTE` behavior.

Taking into account a-priori knowledge about the behavior of the samples in our dataset and observations during the analysis, we defined rules that label a total of 31 distinct behaviors.

Install. We detect modifications to known autostart locations (`AUTO_START`) to recognize when a sample installs on the system.

Networking. We label network behavior based on protocols and port numbers. We label any data flow to the network over port 25 as `SPAM`. Furthermore, we distinguish between `HTTP_REQUEST`, `HTTPS_REQUEST`, `DNS_QUERY`, general `TCP_TRAFFIC` and `UDP_TRAFFIC` as well `LOCAL_CONNECTION` from and to localhost. Finally, `OPEN_PORT` indicates that the malware listens on a local port.

Download & Execute. This describes the updating functionality of a malware sample by labeling any data flow from the network to files that are then executed (`DOWNLOAD_EXECUTE`) or to memory sections that are injected into foreign processes (`DOWNLOAD_INJECT`).

Information Stealing. We currently detect four information-stealing behaviors. `FTP_CREDENTIALS` and `EMAIL_HARVESTING` indicate that the malware harvests FTP credentials and email addresses, respectively, from the filesystem and registry. `BITCOIN_WALLET` indicates that the malware searches for a Bitcoin wallet to steal digital currency. Finally, `INTERNET_HISTORY` is detected when the malware accesses the user's browsing history.

Fake AV. Fake AV software modifies system settings to simulate system instability and persuade the victim to pay for additional software that fixes these "problems". We detect this as modifications to the registry that disable the task manager (`DISABLE_TASKMGR`) or hide items in the start menu (`HIDE_STARTMENU`), as well as enumeration of the file system and setting all file attributes to hidden (`HIDE_FILES`).

Browser Hijacking. We detect this type of behavior when a malware changes Internet Explorer's or Firefox's proxy settings, (`IE_PROXY_SETTINGS` and `FIREFOX_SETTINGS`).

Anti AV. We detect behavior that disables system call hooks commonly used by AV software by restoring the System Service Dispatch Table rom the disk image of the Windows kernel (`RESTORE_SSDT`).

AutoRun. We detect the use of the `AUTO_RUN` feature as modifications to `autorun.inf`, changes to the AutoRun settings in the registry and the enumeration and spreading to external drives.

Lowering Security Settings. We currently detect three types of behaviors that lower a system's security settings: the creation of new Windows firewall's rules or attempts to disable it completely (FIREWALL_SETTINGS), registry modifications that disable Internet Explorer's phishing filter (IE_SECURITY_SETTINGS), and changes to a system security policy that classifies executables as low risk file types when downloading them from the Internet or opening email attachments (CHANGE_SECURITY_POLICIES).

Miscellaneous. We also detect a number of simple behaviors that have self-explanatory labels: INJECT_CODE, START_SERVICE, EXECUTE_TEMP_FILE, ENUMERATE_PROCESSES and DOWNLOAD_FILE.

Unpacking. To identify a malware's unpacking code (UNPACKER), we do not rely on behavior rules as we do for other labels. Instead, we assume that all code found in the original malware binary *before unpacking* is part of the unpacking behavior. The reason is that malware authors use packing to hide as much as possible of their software from analysis and detection: Thus, all other functionality is typically found inside the packing layer.

The first column of Tab. 2 shows the percentage of a sample's overall code that is tagged with any behavior (labeled or unlabeled). This is all the code that is responsible for *any* observed behavior, and is a measure of the code coverage of our dynamic analysis. Overall coverage is relatively low, which confirms the difficulty of performing a complete dynamic analysis. The second column show the percentage that is tagged with a labeled behavior; That is, code to which we were able to attribute a high-level purpose. Except for one outlier (Dapato, at 40.9%) the labeled code is on average $73.4\% - 99.91\%$ of the total tagged code. This shows that BEAGLE was able to assign most executed code to a functional component.

5.7 Behavior Evolution

With the techniques discussed in §4.4, BEAGLE is able to monitor the evolution of each of the detected behaviors across successive malware versions. For each functional components that implements a behavior, BEAGLE can produce results similar to those presented for the overall malware code in Table 2 and in Figures 2 and 3. Due to space limitations, we can present here only a small sample of these results. To present the evolution of behaviors, we focus on the similarity (as defined in Eq. (1)) between each version of the behavior and the *reference behavior*. As discussed in Section 4.4, the reference behavior is the largest implementation of a behavior by number of basic blocks, across a malware's versions. While this does not provide a complete picture of the code's evolution, it gives an idea of how each behavior grows towards its largest implementation (i.e., the reference behavior).

Fig. 5 shows the similarity over time of each behavior found in ZeuS (3rd variant) against the respective reference behavior. This shows the contribution of each behavior to the overall changes. Interestingly, in this family as well as in other families, we notice very limited code change overall (first plot). However, the breakdown reveals some significant changes towards the end of the observation window, where behaviors such as TCP_CONNECTION, DOWNLOAD_INJECT and HTTP_REQUEST change their similarity with respect to the reference.

A more compact representation of the behavior "variability" is exemplified in Fig. 5, which shows the boxplot distribution of the similarity of each behavior against the respective reference behavior. In the family under examination, which is Gamarue, behaviors such as DOWNLOAD_EXECUTE, UDP_TRAFFIC, and DOWNLOAD_FILE almost never change, except for some outliers (empty circles). Other behaviors, instead, exhibit more variance, which means that their

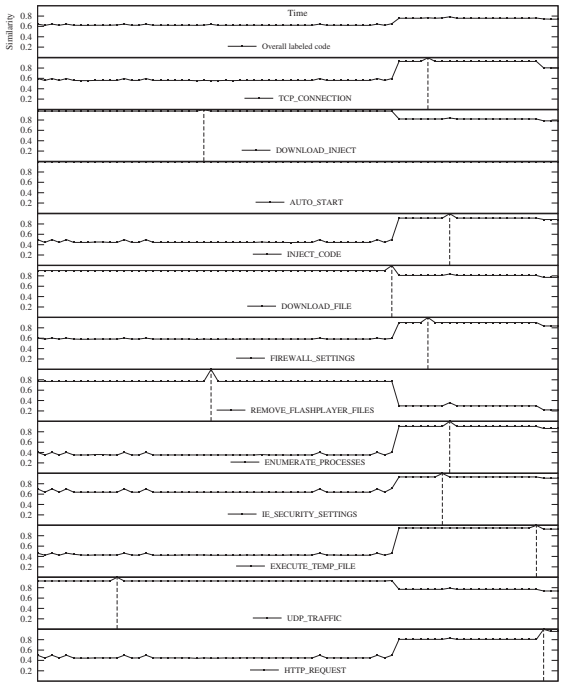


Figure 5: ZeuS (3rd variant): Similarity over time of each behavior against the respective reference behavior. The first timeline is the overall code similarity with respect to the first sample of that family. This plot shows how the overall changes are broke down into changes in the single behaviors. We found analogous patterns also in Gamarue (omitted for space limits).

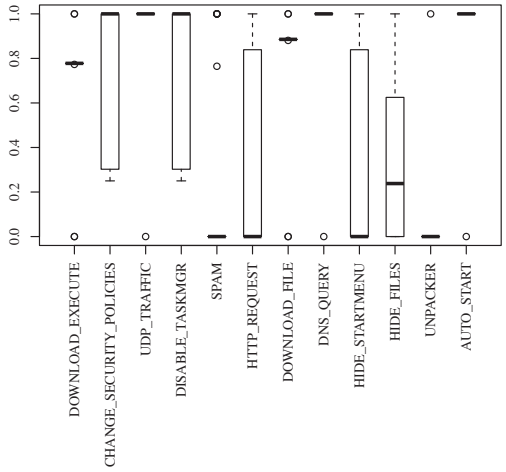


Figure 6: Gamarue: Distribution of the similarity of each behavior against the respective reference behavior. Each box marks the 0%-, 25%- and 75%-, 100%-quantiles, and the median. The circles indicate the outliers.

code is changed often, corresponding to a proportionally larger development effort.

5.8 Lines of Malicious Code

Throughout our evaluation, we have used basic blocks as the unit of measurement for code, whereas it would be more useful to quantify the malware development effort in terms of lines of malicious code. Unfortunately, directly measuring the Lines of

Code (LoC) would require the malware's source code, which is typically unavailable. Nonetheless, we would like to provide a rough estimate of the amount of source code that may correspond to the observed changes. For this, we attempt to estimate a range of possible values for the ratio of basic blocks to LoC in malware samples. Clearly, factors such as the compiler, compiler options and programming paradigms can significantly influence such ratio, thus our estimate is not generalizable outside the scope of our dataset.

We obtained the source code of 150 malware samples of various kinds (e.g., bots, worms, spyware), including the leaked ZeuS source code, and a corresponding executable binary. Apart from ZeuS, none of the families in our dataset are represented in these 150 samples. We then calculated the number of lines of C/C++ source code (LoC), using `cloc`, and the number of basic blocks, using BEAGLE's binary comparison submodule—we excluded the blocks that belong to whitelisted fingerprints. Within this dataset, we found that the ratio between basic blocks and LoC ranges between 50 and 150 blocks per LoC, and around 14.64 for ZeuS. However, one third of the 150 samples that we analyze exhibit a ratio very close to 50 basic blocks per LoC, which we take as a conservative lower bound.

Given these estimates, in the case of ZeuS, the average amount of new code is around 140–280 lines of code, with peaks up to 9,000. Since the Zeus samples in our dataset are closely related to the source code used to estimate this ratio, we consider this a relatively reliable estimate. For the remaining families, with a rough estimate using a ratio of 50, we notice an average amount of *new* code around 100–300 LoC per update cycle, with peaks up to 4,600–9,000. These are just estimates, but they give an overall idea of the significant development effort behind the evolution of malware.

5.9 Conclusive Remarks

BEAGLE revealed that, within our observation window, some families are much more actively developed than others. For instance, GenericTrojan, Llac and ZeuS (3rd variant) have a remarkable amount of new code added. A wider observation window may obviously unveil other "spikes" of development efforts (e.g., new code). As discussed in §5.8, for some families such as ZeuS, we can also give rough estimates of these quantities in lines of source code.

BEAGLE is also able to tell whether and how such changes target each individual behavior. For instance, some behaviors in certain families almost never change (e.g., `UDP_TRAFFIC` or `DOWNLOAD_FILE` in Gamarue), whereas other behaviors (e.g., `HIDE_STARTMENU` or `HTTP_REQUEST` in Gamarue) change over time. In some cases, such as ZeuS (3 rd variant), the overall development effort appears constant, and relatively low. BEAGLE's more focused analysis of the evolution of individual components of this malware, however, reveals that some behaviors undergo significant changes.

6. LIMITATIONS AND FUTURE WORK

Our results demonstrate that BEAGLE is able to provide insight on the real-world evolution of malware samples. However, malware authors could take steps to defeat our system. First of all, the simple unpacking techniques used by BEAGLE could be defeated by using more advanced approaches such as multi-layer or emulation-based packing. For this, BEAGLE could be extended with advanced unpacking approaches [26, 33]. A further problem is that malware analyzed by BEAGLE may be able to detect that it is running in an analysis sandbox, and refuse to update. In recent years, a number of techniques have been proposed to attempt to detect [7, 24] or analyze [12, 19] evasive malware.

Even in the absence of evasion, BEAGLE's dynamic analysis component suffers from limited code coverage. This problem is to some extent alleviated by the fact that we combine information on a malware sample from a large number of executions over a period of months. None-the-less, behavior that is never observed in our sandbox cannot be analyzed. As an example, in our current experiments our observation of the `BITCOIN_WALLET` behavior is incomplete because the analysis environment does not include a Bitcoin wallet for the malware to steal.

BEAGLE can identify and measure the evolution of a malware's functional components. However, it cannot tell us anything more about the semantics of the code changes it detects: This task is currently left to a human analyst. The next logical step is to develop patch analysis techniques to attempt to automatically understand how the update of a component changes its functionality.

7. RELATED WORK

The techniques we apply to measure the similarity between two versions of the same malware family are related to the field of software similarity and classification (as well as plagiarism and theft detection). We refer the reader to [10], which presents a comprehensive review of the existing methods to analyze the similarity of non-binary and binary code, including malicious code.

BinHunt [14] was developed to facilitate patch analysis by accurately identifying which functions have been modified and to what extent. For this, the authors compare the program call graphs and control flow graphs, using symbolic execution and theorem proving to prove that two basic blocks are functionally equivalent, despite transformations such as register substitution and instruction reordering. BinHunt however does not work on packed code, and its efficiency decreases as the amount of code differences increases. BEAGLE's binary comparison component is also based on control flow graphs, and is robust to some code transformations such as basic block and instruction re-ordering and register substitution. While less precise, our approach is fast and scalable and can also be used to contrast a binary against multiple others. Bitshred [18] uses n-grams over binary code and bloom filters to efficiently and scalably locate re-used code in large datasets. Since it relies on raw byte sequences, however, this approach is less robust to syntactic changes in binary code.

The work most closely related to our own is Reanimator [28] which can identify the code responsible for a malware behavior observed in dynamic analysis at instruction granularity and uses CFG fingerprints [23] as signatures to detect the presence of this code in other malware. Our techniques for mapping behavior to code are less precise, and produce results at function granularity. The advantage is that we do not require (extremely slow) instruction-level logging, and are able to apply our techniques to a larger dataset.

Roberts [30] presents some early, high-level insights on the malware development life cycle, largely based on manual malware analysis efforts. More recently, in work concurrent to our own, Rossow et al. [31] performed a large scale analysis of malware downloaders and the C&C infrastructure they rely on.

8. CONCLUSIONS

Understanding the mechanics and economics of malware evolution over time is an interesting and challenging research problem with practical applications. We proposed an automated approach to associate observed behaviors of a malware binary with the components that implement them, and to measure the evolution of each component across malware versions. To the best of our knowledge, no previous research has automatically monitored how malware components change over time.

Our system can observe the overall evolution of a malware sample and of its individual functional components. This led us to interesting insights on the development efforts of malicious code.

Our measurements confirmed commonly held beliefs (e.g., that the malware industry is partly driven by AV advances), but gave also novel and interesting insights. For instance, we observed that most malware writers reuse significant portions of code, but that this varies wildly by family, with significant "spikes" of software development in short timespans. We were also able to distinguish between behaviors that never change in a certain family, and others being constantly developed; In some cases, spikes of development of a malware component are not visible in the overall evolution of the malware sample, but are revealed by the analysis of individual behaviors.

BEAGLE proved to be useful both to build the "big picture" of how and when self-updating malware change, and to guide a malware analysts to the most interesting portions of code (i.e., parts that have changed significantly between two successive versions).

9. ACKNOWLEDGMENTS

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement no 257007.

10. REFERENCES

[1] BinDiff. http://www.zynamics.com/bindiff.html.

[2] Microsoft Malware Protection Center. http://www.microsoft.com/security/portal/Threat/Views.aspx.

[3] ZeuS Tracker. http://zeustracker.abuse.ch/.

[4] Symantec Report on Rogue Security Software. Symantec White Paper, October 2009.

[5] PandaLabs Annual Report. http://press.pandasecurity.com/wp-content/uploads/2012/01/Annual-Report-PandaLabs-2011.pdf, 2011.

[6] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario. Automated Classification and Analysis of Internet Malware. In *RAID*, 2007.

[7] D. Balzarotti, M. Cova, C. Karlberger, C. Kruegel, E. Kirda, and G. Vigna. Efficient Detection of Split Personalities in Malware. In *NDSS*, 2010.

[8] U. Bayer, C. Kruegel, and E. Kirda. TTAnalyze: A Tool for Analyzing Malware. In *EICAR Annual Conf.*, 2006.

[9] U. Bayer, P. Milani Comparetti, C. Hlauscheck, C. Kruegel, and E. Kirda. Scalable, Behavior-Based Malware Clustering. In *NDSS*, 2009.

[10] S. Cesare and Y. Xiang. *Software Similarity and Classification*. Springer-Verlag, 2012.

[11] M. Christodorescu, S. Jha, and C. Kruegel. Mining Specifications of Malicious Behavior. In *European Software Engineering Conf. and ACM Symposium on the Foundations of Software Engineering (ESEC-FSE)*, 2007.

[12] A. Dinaburg, P. Royal, M. I. Sharif, and W. Lee. Ether: Malware Analysis via Hardware Virtualization Extensions. In *CCS*, 2008.

[13] H. Flake. Structural Comparison of Executable Objects. In *DIMVA*, 2004.

[14] D. Gao, M. K. Reiter, and D. Song. BinHunt: Automatically Finding Semantic Differences in Binary Programs. In *ICICS*, 2008.

[15] M. Hayes, A. Walenstein, and A. Lakhotia. Evaluation of Malware Phylogeny Modelling Systems Using Automated Variant Generation. *Journal in Computer Virology*, 5(4):335–343, 2009.

[16] G. Jacob, H. Debar, and E. Filiol. Malware Behavioral Detection by Attribute-Automata using Abstraction from Platform and Language. In *RAID*, 2009.

[17] G. Jacob, R. Hund, C. Kruegel, and T. Holz. JACKSTRAWS: Picking Command and Control Connections from Bot Traffic. In *USENIX Security Symposium*, 2011.

[18] J. Jang, D. Brumley, and S. Venkataraman. BitShred: Feature Hashing Malware for Scalable Triage and Semantic Analysis. In *CCS*, 2011.

[19] N. M. Johnson, J. Caballero, K. Z. Chen, S. McCamant, P. Poosankam, D. Reynaud, and D. Song. Differential Slicing: Identifying Causal Execution Differences for Security Applications. In *SSP*, 2011.

[20] M. Karim, A. Walenstein, A. Lakhotia, and L. Parida. Malware Phylogeny Generation Using Permutations of Code. *Journal in Computer Virology*, 1(1):13–23, 2005.

[21] B. Krebs. Criminal Classifieds: Malware Writers Wanted. https://krebsonsecurity.com/2011/06/criminal-classifieds-malware-writers-wanted/, June 2011.

[22] B. Krebs. Banking on Badb in the Underweb. https://krebsonsecurity.com/2012/03/banking-on-badb-in-the-underweb/, Mar. 2012.

[23] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic Worm Detection Using Structural Information of Executables. In *RAID*, 2005.

[24] M. Lindorfer, C. Kolbitsch, and P. Milani Comparetti. Detecting Environment-Sensitive Malware. In *RAID*, 2011.

[25] F. Maggi, A. Bellini, G. Salvaneschi, and S. Zanero. Finding Non-trivial Malware Naming Inconsistencies. In *ICISS*, 2011.

[26] L. Martignoni, M. Christodorescu, and S. Jha. OmniUnpack: Fast, Generic, and Safe Unpacking of Malware. In *ACSAC*, 2007.

[27] L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. C. Mitchell. A Layered Architecture for Detecting Malicious Behaviors. In *RAID*, 2008.

[28] P. Milani Comparetti, G. Salvaneschi, E. Kirda, C. Kolbitsch, C. Kruegel, and S. Zanero. Identifying Dormant Functionality in Malware Programs. In *SSP*, 2010.

[29] R. Perdisci, A. Lanzi, and W. Lee. Classification of Packed Executables for Accurate Computer Virus Detection. *Pattern Recognition Letters*, 29(14):1941–1946, 2008.

[30] R. Roberts. Malware Development Life Cycle. *Virus Bulletin Conf.*, (October), 2008.

[31] C. Rossow, C. J. Dietrich, and H. Bos. Large-Scale Analysis of Malware Downloaders. In *DIMVA*, 2012.

[32] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware. In *ACSAC*, 2006.

[33] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Automatic Reverse Engineering of Malware Emulators. In *SSP*, 2009.

[34] B. Stone-Gross, T. Holz, G. Stringhini, and G. Vigna. The Underground Economy of Spam: A Botmaster's Perspective of Coordinating Large-Scale Spam Campaigns. In *USENIX LEET*, 2011.

[35] D. Thomas, K. an Nicol. The Koobface Botnet and the Rise of Social Malware. In *Conf. On Malicious and Unwanted Software (MALWARE)*, 2010.

[36] Y.-M. Wang, R. Roussev, C. Verbowski, A. Johnson, M.-W. Wu, Y. Huang, and S.-Y. Kuo. Gatekeeper: Monitoring Auto-Start Extensibility Points (ASEPs) for Spyware Management. In *USENIX LISA*, 2004.

Generalized Vulnerability Extrapolation using Abstract Syntax Trees

Fabian Yamaguchi
University of Göttingen
Göttingen, Germany

Markus Lottmann
Technische Universität Berlin
Berlin, Germany

Konrad Rieck
University of Göttingen
Göttingen, Germany

ABSTRACT

The discovery of vulnerabilities in source code is a key for securing computer systems. While specific types of security flaws can be identified automatically, in the general case the process of finding vulnerabilities cannot be automated and vulnerabilities are mainly discovered by manual analysis. In this paper, we propose a method for assisting a security analyst during auditing of source code. Our method proceeds by extracting abstract syntax trees from the code and determining structural patterns in these trees, such that each function in the code can be described as a mixture of these patterns. This representation enables us to decompose a known vulnerability and extrapolate it to a code base, such that functions potentially suffering from the same flaw can be suggested to the analyst. We evaluate our method on the source code of four popular open-source projects: LibTIFF, FFmpeg, Pidgin and Asterisk. For three of these projects, we are able to identify zero-day vulnerabilities by inspecting only a small fraction of the code bases.

1. INTRODUCTION

The security of computer systems critically depends on the quality of its underlying code. Even minor flaws in a code base can severely undermine the security of a computer system and make it an easy victim for attackers. There exist several examples of vulnerabilities that have led to security incidents and the proliferation of malicious code in the past [e.g. 21, 26]. A drastic case is the malware Stuxnet [7] that featured code for exploiting four unknown vulnerabilities in the Windows operating system, rendering conventional defense techniques ineffective in practice.

The discovery of vulnerabilities in source code is a central issue of computer security. Unfortunately, the process of finding vulnerabilities cannot be automated in the general case. According to Rice's theorem a computer program is unable to generally decide whether another program contains vulnerable code [10]. Consequently, security research has focused on devising methods for identifying specific types of vulnerabilities.

Several approaches have been proposed that statically identify patterns of specific vulnerabilities [e.g., 4, 18, 28, 32], such as the use of certain insecure functions. Moreover, concepts from the area of software verification have been successfully adapted for tracking vulnerabilities, for example, in form of fuzz testing [27], taint analysis [22] and symbolic execution [1, 8]. Many of these approaches, however, are limited to specific conditions and types of vulnerabilities. The discovery of vulnerabilities in practice still mainly rests on tedious manual auditing that requires considerable time and expertise.

In this paper, we propose a method for assisting a security analyst during auditing of source code. Instead of striving for an automated solution, we aim at rendering manual auditing more effective by guiding the search for vulnerabilities. Based on the idea of *vulnerability extrapolation* [33], our method proceeds by extracting abstract syntax trees from the source code and determining structural patterns in these trees, such that each function in the code can be described as a mixture of the extracted patterns. The patterns contain subtrees with nodes corresponding to types, functions and syntactical constructs of the code base. This representation enables our method to decompose a known vulnerability and to suggest code with similar properties—potentially suffering from the same flaw—to the analyst for auditing.

We evaluate the efficacy of our method using the source code of four popular open-source projects: LibTIFF, FFmpeg, Pidgin and Asterisk. We first demonstrate in an quantitative evaluation how functions are decomposed into structural patterns and how similar code can be identified automatically. In a controlled experiment we are able to narrow the search for a given vulnerability to 8.7% of the code base and consistently outperform non-structured approaches for vulnerability extrapolation. We also study the discovery of real vulnerabilities in a qualitative evaluation, where we are able to discover 10 zero-day vulnerabilities in the source code of the four open-source projects.

In summary, we make the following contributions:

- *Generalized vulnerability extrapolation:* We present a general approach to the extrapolation of vulnerabilities, allowing both the content and structure of code to be considered for finding similar flaws in a code base.
- *Structural comparison of code:* We present a method for robust extraction and analysis of abstract syntax trees that allows for automatic comparison of code with respect to structural patterns.

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

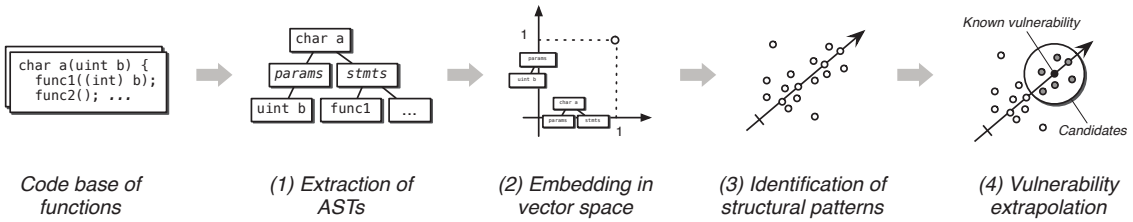


Figure 1: Schematic overview of our method for vulnerability extrapolation.

- *Evaluation and cases studies:* We study the capabilities of our method in different empirical experiments, where we identify real zero-day vulnerabilities in popular open-source projects.

The rest of this paper is structured as follows: our method for vulnerability discovery is introduced in Section 2 and its evaluation is presented in Section 3. Limitations and related work are discussed in Section 4 and 5, respectively. Section 6 concludes this paper.

2. VULNERABILITY EXTRAPOLATION

The concept of vulnerability extrapolation builds on the observation that source code often contains several vulnerabilities linked to the same flawed programming patterns, such as missing checks before or after function calls. Given a known vulnerability, it is thus often possible to discover previously unknown vulnerabilities by finding functions sharing similar code structure.

In practice, such extrapolation of vulnerabilities is attractive for two reasons: First, it is a general approach that is not limited to any specific vulnerability type. Second, the extrapolation does not hinge on any involved analysis machinery: a robust parser and an initial vulnerability are sufficient for starting an analysis. However, assessing the similarity of code is a challenging task, as it requires analyzing and comparing structured objects, such as subtrees of syntax trees. Previous work has thus only considered flat representations, such as function and type names, for extrapolating vulnerabilities [see 33].

To tackle the challenge of structured data, our method combines concepts from static analysis, robust parsing and machine learning. It proceeds in four steps that are illustrated in Figure 1 and described in the following:

1. *Extraction of abstract syntax trees.* In the first step, abstract syntax trees (AST) are extracted for all functions of the code base using a robust parser. This parser is based on the concept of island grammars [20] and capable of extracting syntax trees from C/C++ source code even without a working build environment (Section 2.1).

2. *Embedding in a vector space.* The abstract syntax trees of the functions are then embedded in a vector space, such that techniques from machine learning can be applied to analyze the code. The embedding is accomplished by disregarding irrelevant nodes in the trees and representing each function as a vector of contained subtrees (Section 2.2).

3. *Identification of structural patterns.* Based on the vectorial representation, structural patterns are identified in the code using the technique of latent semantic analysis [5]. This analysis technique determines dominant directions (structural patterns) in the vector space corresponding to combinations of AST subtrees frequently occurring in the code base (Section 2.3).

4. *Vulnerability Extrapolation.* Finally, the functions of the code base are described as mixtures of the identified structural patterns. This representation enables identifying code similar to a known vulnerability by finding functions with a similar mixture of structural patterns (Section 2.4).

In the following, these four steps are described in detail and the required theoretical and technical background is presented where necessary.

2.1 Robust AST Extraction

Determining arbitrary structural patterns in code demands a fine grained representation of code, similar in precision to the abstract syntax trees (AST) generated by compilers. Obtaining these trees directly from a compiler is only possible if a working build environment is available. Unfortunately, constructing such an environment is often non-trivial in practice, as all dependencies of the code including correct versions of build tools and header files need to be available. Finally, when analyzing legacy code, parts of the code may simply not be available anymore.

As a remedy, we employ a robust parser for C/C++ based on the concept of *island grammars* [20]. This parser allows for extracting ASTs from individual source files. In contrast to parsers integrated with a compiler, this parser does not aim to validate the syntax of the code it processes. Instead, the objective is to extract as much information from the code as possible, assuming that it is syntactically valid in the first place.

Our parser is based on a single grammar definition for the ANTLR parser generator [23] and publicly available[1]. The parser outputs ASTs in a serialized text format as shown in Figure 2. This serialized format is well suited for subsequent processing and provides generic access to the structure of the parsed code. A graphical version of the corresponding tree is presented in Figure 3.

For our analysis we distinguish between different types of nodes in the syntax tree. We refer to all nodes associated with parameter types, declaration types and function calls as *API nodes* (dashed in Figure 3), as they define how the code interfaces with other functions and libraries. Moreover, we denote all nodes describing syntactical elements as *syntax nodes* (dotted in Figure 3).

[1]http://codeexploration.blogspot.de/

```
1 int foo(int y)
2 {
3     int n = bar(y);
4
5     if (n == 0)
6         return 1;
7
8     return (n + y);
9 }
```

(a) Exemplary C function

```
# type  depth  value1   value2
func    0       int     foo
params  1
param   2       int     y
stmts   1
decl    2       int     n
op      2       =
call    3       bar
arg     4       y
if      2       (n == 0)
cond    3       n == 0
op      4       ==
stmts   3
return  4       1
return  2       (n + y)
op      3       +
```

(b) Serialized AST

Figure 2: Example of a C function and a serialized AST

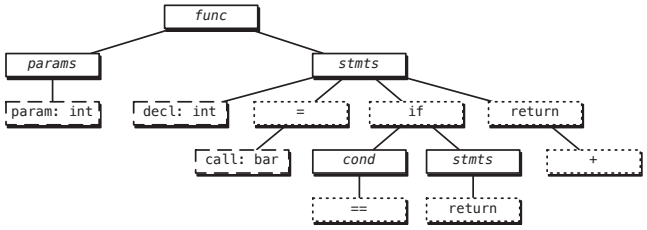


Figure 3: Abstract syntax tree with API nodes (dashed) and syntax nodes (dotted).

2.2 Embedding of ASTs in a Vector Space

Abstract syntax trees offer a rich source of information for extraction of code patterns. However, machine learning techniques cannot be applied directly to this type of data, as they usually operate on numerical vectors.

To address this problem, a suitable map is required, allowing ASTs to be transformed into vectors. This map needs to capture the structure and content of the trees, and thus is crucial for the success of vulnerability extrapolation. To construct this map, we describe the AST of each functions in our code base using a set of subtrees $\mathcal{S}$. In particular, we experiment with the following three definitions of the set:

1. *API nodes.* We consider only flat function and type names [see 33]. The set $\mathcal{S}$ simply consists of all individual API nodes found in the ASTs of the code base. All other nodes are ignored.

2. *API subtrees.* The set $\mathcal{S}$ is defined as all subtrees of depth D in the code base that contain at least one API node. The subtrees are generalized by replacing all non-API nodes with placeholders (empty nodes).

3. *API/S subtrees.* The set $\mathcal{S}$ consists of all subtrees of depth D containing at least one API or syntax node. Again, all non-API and non-syntax nodes are replaced by placeholders (empty nodes).

Depending on this definition of $\mathcal{S}$, we obtain different views on the functions of the code base. If we consider API nodes only, our characterization is shallow and we only capture the interfacing of the functions. If we choose API subtrees as the set $\mathcal{S}$, we describe the functions in terms of API nodes and the structural context these nodes occur in. Finally, if we consider API/S subtrees, we obtain a view on our code base that reflects API usage as well as the occurrences of syntactical elements in the functions. In the following we fix the depth of subtrees to $D = 3$ in all experiments, as this setting provides a good balance between a shallow representation and overly complex subtrees.

Based on the set $\mathcal{S}$ we can define a map ϕ that embeds an AST x in a vector space, where each dimension is associated with one element of $\mathcal{S}$. Formally, this map is given by

$$\phi : \mathcal{X} \longmapsto \mathbb{R}^{|\mathcal{S}|}, \quad \phi(x) \longrightarrow \big(\#(s,x) \cdot w_s\big)_{s \in \mathcal{S}}$$

where $\mathcal{X}$ refers to all ASTs of functions in our code base and $\#(s,x)$ returns the number of occurrences of the subtree $s \in \mathcal{S}$ in x. For convenience and later processing, we store the vectors of all ASTs in our code base in a matrix M, where one cell of the matrix is defined as $M_{s,x} = \#(s,x) \cdot w_s$.

The term w_s in the map ϕ corresponds to a TF-IDF weighting. This weighting ensures that subtrees occurring very frequently in the code base have little effect when assessing function similarity. Furthermore, it removes the bias towards longer functions, which can contain similar subtrees to a lot of different functions but are not particularly similar to any of them. A detailed description of this weighing scheme is given by Salton and McGill [25].

Let us, as an example, consider the AST given in Figure 3 and the set $\mathcal{S}$ of API nodes. The tree x contains only three API nodes, namely `param: int`, `decl: int` and `call: bar`. As a result, the corresponding three dimensions in the vector $\phi(x)$ are non-zero, whereas all other dimensions are zero. The vector space constructed by the map ϕ may contain hundred thousands of dimensions, yet the vectors are extremely sparse. This sparsity can be exploited for efficiently storing and comparing the vectors in practice.

2.3 Identification of Structural Patterns

By calculating distances between vectors, the representation obtained in the previous step already allows functions to be compared in terms of the subtrees they share. However, we cannot yet compare functions with respect to more involved patterns. For example, the code base of a server application may contain functions related to network communication, message parsing and thread scheduling. In this setting, it would be better to compare the functions with respect to these functionalities rather than looking at the plain subtrees of the ASTs.

Fortunately, we can adapt the technique of *latent semantic analysis* [5] to solve this problem. Latent semantic analysis is a classic technique of natural language processing that is used for identifying topics in text documents. Each topic is represented by a vector of related words. In our setting these topics correspond to types of functionality in the code base and the respective vectors are associated with subtrees related to these functionalities.

Latent semantic analysis identifies topics by determining dominant directions in the vector space, that is, subtrees frequently occurring together in ASTs of our code base. We refer to these directions of related subtrees as *structural patterns*. By projecting the original vectors on the identified directions, one obtains a low-dimensional representation of the data. Each AST of a function is described as a mixture of the structural patterns. For example, a function related to communication and parsing is represented as a mixture of patterns corresponding to these types of functionality.

Formally, latent semantic analysis seeks d orthogonal directions in the vector space that capture as much of the variance inside the data as possible. One technical way to obtain these d directions is by performing a singular value decomposition (SVD) of the matrix M. That is, M is decomposed into three matrices U, Σ and V as follows

$$M \approx U\Sigma V^T = \begin{pmatrix} \leftarrow u_1 \rightarrow \\ \leftarrow u_2 \rightarrow \\ \vdots \\ \leftarrow u_{|\mathcal{S}|} \rightarrow \end{pmatrix} \begin{pmatrix} \sigma_1 & 0 & \dots & 0 \\ 0 & \sigma_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \sigma_d \end{pmatrix} \begin{pmatrix} \leftarrow v_1 \rightarrow \\ \leftarrow v_2 \rightarrow \\ \vdots \\ \leftarrow v_{|\mathcal{X}|} \rightarrow \end{pmatrix}^T .$$

The decomposition provides a wealth of information and contains the projected vectors as well as the structural patterns identified in the matrix M.

1. The d columns of the unitary matrix U correspond to the dominant directions in the vector space and define the d structural patterns of subtrees identified in the code base.

2. The diagonal matrix Σ contains the singular values of M. The values indicate the variances of the directions and allow us to assess the importance of the d structural patterns.

3. The rows of V contain the projected representations of the embedded ASTs, where each AST is described by a mixture of the d structural patterns contained in the matrix U.

As we will see in the following, these three matrices provide the basis for extrapolation of vulnerabilities and conclude the rather theoretical presentation of our method.

2.4 Extrapolation of Vulnerabilities

Once the decomposition has been calculated, which takes a fraction of the time required for code parsing, an analyst is able to access the information encoded in the three matrices. In particular, the following three activities can be performed instantaneously to assist code auditing.

- *Vulnerability extrapolation.* The rows of the matrix V describe all functions as mixtures of structural patterns. Finding structurally similar functions is thus as simple as comparing the rows of V using a suitable measure, such as the cosine distance [25]. This operation forms the basis for the extrapolation of vulnerabilities. Clearly, there is no guarantee that functions with similar structure are plagued by the same vulnerabilities, however, examples presented in Section 3 provide some evidence for this correspondence.

- *Code base decomposition.* At the beginning of an audit, little is known about the overall structure of the code base. In this setting, the matrix U storing the most prevalent structural patterns in its columns gives important insight into the structure of the code base. This information can be used to uncover major clusters of similar functions, such as sets of functions employing similar programming patterns. This allows an analyst to select interesting parts of the code early in the audit and concentrate on promising functions first.

- *Detection of unusual functions.* Finally, the representation of functions in terms of structural patterns is fully transparent, allowing an analyst to discover the most prominent patterns used in any particular function by comparing rows of V with columns of U. This enables determining deviations from common programming patterns by analyzing differences in the representations of functions. For example, it might be interesting to audit a function related to message parsing that deviates from other such functions by also sharing structural patterns with network communication.

3. EVALUATION

We proceed to evaluate our method with real source code. In particular, we are interested in studying the ability of our method to assess the similarity of code and to identify potentially vulnerable functions in practice. We first conduct a *quantitative evaluation*, where we apply our method in a controlled experiment on different code bases. We then present a *qualitative evaluation* and examine the extrapolation of real vulnerabilities in two case studies.

For the evaluation we consider four popular open-source projects, namely LibTIFF, FFmpeg, Pidgin and Asterisk. For each of these projects we pick one known vulnerability as a starting point for the vulnerability extrapolation and proceed to manually label *candidate functions* which should be reviewed for the same type of vulnerability.

In the following, we describe the code bases of these projects and the choice of candidate functions in detail:

1. LibTIFF (`http://www.libtiff.org`) is a library for processing images in the TIFF format. Its source code covers 1,292 functions and 52,650 lines of code. Version 3.8.1 of the library contains a stack-based buffer overflow in the parsing of TLV elements that allows an attacker to execute arbitrary code using specifically crafted images (CVE-2006-3459). Candidate functions are all parsers for TLV elements.

2. Pidgin (`http://www.pidgin.im`) is a client for instant messaging implementing several communication protocols. The implementation contains 11,505 functions and 272,866 lines of code. Version 2.10.0 of the client contains a vulnerability in the implementation of the AIM protocol (CVE-2011-4601). An attacker is able to remotely crash the client using crafted messages. Candidate functions are all AIM protocol handlers converting incoming binary messages to strings.

3. FFmpeg (`http://www.ffmpeg.org`) is a library for conversion of audio and video streams. Its code base spans 6,941 functions with a total of 298,723 lines of code. A

	API nodes			API subtrees			API/S subtrees		
	75%	90%	100%	75%	90%	100%	75%	90%	100%
Pidgin	0.1	0.36	2.00	0.35	0.22	0.98	0.22	0.67	25.98
LibTIFF	6.35	6.97	7.58	5.65	6.66	7.27	6.49	9.36	17.32
FFmpeg	6.17	8.10	19.61	5.00	8.66	11.09	7.71	15.21	28.35
Asterisk	0.06	10.64	15.29	0.24	10.23	15.54	1.19	16.50	28.45
Average	3.17	6.52	11.12	2.81	6.44	8.72	3.90	10.44	25.03

Table 1: Performance of vulnerability extrapolation in a controlled experiment. The performance is given as amount of code (%) to be audited to find 75%, 90% and 100% of the potentially vulnerable functions.

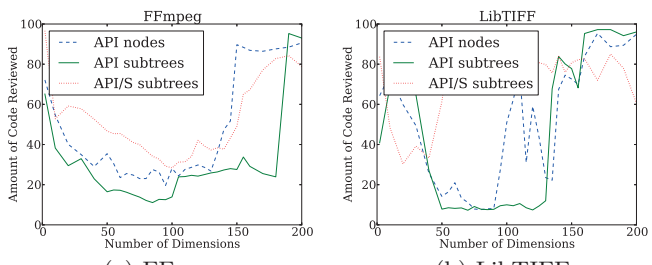
vulnerability has been identified in version 0.6 (CVE-2010-3429). During the decoding of video frames, indices are incorrectly computed, enabling the execution of arbitrary code. Candidate functions are all video decoding routines, which write decoded video frames to a pixel buffer.

4. Asterisk (`http://www.asterisk.org`) is a framework for Voice-over-IP communication. The code base covers 8,155 functions and 283,883 lines of code. Version 1.6.1.0 of the framework contains a vulnerability (CVE-2011-2529), which allows a remote attacker to corrupt memory of the server and to cause a denial of service via a crafted packet. Candidate functions are all functions reading incoming packets from UDP/TCP sockets.

3.1 Quantitative Evaluation

In our first experiment, we study the ability of our method to identify functions sharing similarities with a known vulnerability on the four code bases. To conduct a controlled experiment we thoroughly inspect each code base and manually label all candidate functions, that is, all functions that potentially contain the same vulnerability. Note that this manual analysis process required several weeks of work and can hardly be seen as an alternative to the concept of vulnerability extrapolation.

For each of the four code bases, we apply our method and rank the functions according to the selected target vulnerabilities. We vary the embedding of syntax trees by considering flat API nodes, API subtrees and API/S subtrees (see Section 2.2). Moreover, we compute the ranking for different numbers of structural patterns identified by latent semantic analysis (see Section 2.3). As performance measure we assess the efficacy of the vulnerability extrapolation by measuring the amount of code that needs to be inspected for finding all candidate functions.



(a) FFmpeg (b) LibTIFF

Figure 4: Performance of vulnerability extrapolation in a controlled experiment.

Figure 4 shows the results of this experiment for FFmpeg and LibTIFF, where results for the other two code bases are similar. The API subtrees clearly outperform the other representations of the code and enable narrowing the set of functions to be inspected to 8.7% on average. By contrast, the flat representation of API nodes requires 11.1% of the functions to be reviewed, while for the API/S subtrees even every 4th function (25%) needs to be inspected. Furthermore, Figure 4 also shows that the number of extracted structural patterns is not a critical parameter for vulnerability extrapolation. Our method performs well on all code bases when this number is between 50 to 100 dimensions, despite the fact that FFmpeg contains 6,941 and LibTIFF only 1,292 functions. In the following case studies, we fix this parameter to 70.

Table 1 presents a fine-grained analysis of the performance for each code base, where the amount of code that needs to be audited for revealing 75%, 90% and 100% of the candidate functions is shown. All numbers are expressed in percent of the code base to account for their different sizes. The API subtrees perform best, where 75% of the candidate functions are discovered by reading under 3% of the code bases. In the case of Pidgin and Asterisk, this number further reduces to less than 1% of the entire code base providing a significant advantage over manual auditing.

Nevertheless, the results also show room for improvement, particularly when all candidate functions need to be discovered. In this case, the amount of code to be read reaches 8.7% on average and up to 16% in the worst case. However, even in the worst case the amount of code that needs to be inspected is reduced by 84% and vulnerability extrapolation clearly accelerates manual code auditing in practice.

3.2 Qualitative Evaluation

In a case study with FFmpeg and Pidgin, we now demonstrate the practical merit of vulnerability extrapolation and show how our method plays the key role in identifying eight zero-day vulnerabilities. We have conducted two further studies with Pidgin and Asterisk uncovering two more zero-day vulnerabilities. For the sake of brevity however, we omit these case studies and details of the vulnerabilities here.

3.2.1 *Case study: FFmpeg.*

Flaws in the indexing of arrays are a frequently occurring problem in media libraries. In many cases these vulnerabilities allow attackers to write data to arbitrary locations in memory, an exploit primitive that can often be leveraged for arbitrary code execution. In this case study, we show how a publicly known vulnerability in the video decoder for FLIC media files of FFmpeg (CVE-2010-3429) is used to uncover three further vulnerabilities of this type, two of which were previously unknown. Note that this is the same vulnerabil-

```
1  static int flic_decode_frame_8BPP(AVCodecContext *avctx,
2                                    void *data, int *data_size,
3                                    const uint8_t *buf,
4                                    int buf_size)
5  { [...]
6    signed short line_packets; int y_ptr;
7    [...]
8    pixels = s->frame.data[0];
9    pixel_limit = s->avctx->height * s->frame.linesize[0];
10   frame_size = AV_RL32(&buf[stream_ptr]); [...]
11   frame_size -= 16;
12   /* iterate through the chunks */
13   while ((frame_size > 0) && (num_chunks > 0)) { [...]
14     chunk_type = AV_RL16(&buf[stream_ptr]);
15     stream_ptr += 2;
16     switch (chunk_type) { [...]
17     case FLI_DELTA:
18         y_ptr = 0;
19         compressed_lines = AV_RL16(&buf[stream_ptr]);
20         stream_ptr += 2;
21         while (compressed_lines > 0) {
22             line_packets = AV_RL16(&buf[stream_ptr]);
23             stream_ptr += 2;
24             if ((line_packets & 0xC000) == 0xC000) {
25                 // line skip opcode
26                 line_packets = -line_packets;
27                 y_ptr += line_packets * s->frame.linesize[0];
28             } else if ((line_packets & 0xC000) == 0x4000) {
29                 [...]
30             } else if ((line_packets & 0xC000) == 0x8000) {
31                 // "last byte" opcode
32                 pixels[y_ptr + s->frame.linesize[0] - 1] =
33                     line_packets & 0xff;
34             } else { [...]
35                 y_ptr += s->frame.linesize[0];
36             }
37         }
38         break; [...]
39     }
40   [...]
41   } [...]
42   return buf_size;
43 }
```

```
static void vmd_decode(VmdVideoContext *s)
{
  [...] int frame_x, frame_y;
  int frame_width, frame_height;
  int dp_size;

  frame_x = AV_RL16(&s->buf[6]);
  frame_y = AV_RL16(&s->buf[8]);
  frame_width = AV_RL16(&s->buf[10]) - frame_x + 1;
  frame_height = AV_RL16(&s->buf[12]) - frame_y + 1;

  if ((frame_width == s->avctx->width &&
       frame_height == s->avctx->height) &&
      (frame_x || frame_y)) {
    s->x_off = frame_x;
    s->y_off = frame_y;
  }
  frame_x -= s->x_off;
  frame_y -= s->y_off; [...]
  if (frame_x || frame_y || (frame_width != s->avctx->width) ||
      (frame_height != s->avctx->height)) {
       memcpy(s->frame.data[0], s->prev_frame.data[0],
          s->avctx->height * s->frame.linesize[0]);
  } [...]
  if (s->size >= 0) {
      pb = p;
      meth = *pb++; [...]
      dp = &s->frame.data[0][frame_y * s->frame.linesize[0]
          + frame_x];
      dp_size = s->frame.linesize[0] * s->avctx->height;
      pp = &s->prev_frame.data[0][frame_y *
             s->prev_frame.linesize[0] + frame_x];
      switch (meth) { [...]
      case 2:
        for (i = 0; i < frame_height; i++) {
            memcpy(dp, pb, frame_width);
            pb += frame_width;
            dp += s->frame.linesize[0];
            pp += s->prev_frame.linesize[0];
        } break; [...]
        }
    }
}
```

Figure 5: Original vulnerability in `flic_decode_frame_8BPP` (left) and zero-day vulnerability found in `vmd_decode` (right)

ity extrapolated in our previous work [33], thus allowing the advantages of our improved method to be demonstrated in direct comparison to a non-structured approach.

Original vulnerability. Video decoding in general involves processing image data in form of video frames. These frames contain the encoded pixels as well as meta information of the image, such as width and offset values. Decoders usually proceed by allocating an array and then using the offsets provided in the video frame to populate the array with the image data. In this context, it must be carefully verified that offsets specified by the frame refer to locations within the array. In the case of the decoding routine `flic_decode_frame_8BPP` shown in Figure 5, no verification of this kind is performed allowing an attacker to reference locations outside the pixel array.

The critical write is performed on line 32, where the least significant byte of the user-supplied integer `line_packets` is written to a location relative to the buffer `pixels`. It has been overlooked that the offset is dependent on `y_ptr` and `s->frame.linesize[0]`, both of which can be controlled by an attacker. Due to the loop starting at line 21, it is possible to assign an arbitrary value to `y_ptr` independent of the last value stored in `line_packets` and no check is performed to verify whether the offset remains within the buffer.

Extrapolation. We proceed by using our method to generate the ranking shown in Table 2. This ranking contains the 30 most similar functions to the vulnerable function `flic_decode_frame_8BPP` selected from a total of 6,941 functions in the FFmpeg code base. Candidate functions for the vulnerabilities are depicted with light shading, discovered vulnerabilities are indicated by dark shading.

First, we note that 20 out of 30 functions are candidate functions, i.e., they are decoders performing a write operation to a pixel buffer. Furthermore, no irrelevant functions are part of the first 13 results and we can spot four vulnerable functions in the ranking corresponding to the following three vulnerabilities:

1. The function `flic_decode_frame_15_16BPP` is located in the same file as the original vulnerability and likewise processes FLIC video frames. The function has been found by FFmpeg developers to contain a similar vulnerability, which was patched along with the original flaw. Our method returns a similarity of 98%.

2. The function `vmd_decode` depicted in Figure 5 contains the vulnerability discovered in [33]. The function proceeds by employing the same API functions used in the original vulnerability to read frame meta data on line 7 to 10 and then uses these values to calculate an incorrect index into the pixel buffer on line 28. Our method returns a similarity of 89%, leading us almost instantly to this vulnerability.

```
1  static void vqa_decode_chunk(VqaContext *s)
2  {
3    [...]
4    int lobytes = 0;
5    int hibytes = s->decode_buffer_size / 2; [...]
6    for (y = 0; y < s->frame.linesize[0] * s->height;
7      y += s->frame.linesize[0] * s->vector_height){
8      for (x = y; x < y + s->width; x += 4, lobytes++, hibytes++)
9        {
10       pixel_ptr = x;
11       /* get the vector index, the method for
12          which varies according to
13        * VQA file version */
14       switch (s->vqa_version) {
15       case 1: [...]
16       case 2:
17       lobyte = s->decode_buffer[lobytes];
18       hibyte = s->decode_buffer[hibytes];
19       vector_index = (hibyte << 8) | lobyte;
20       vector_index <<= index_shift;
21       lines = s->vector_height;
22       break;
23       case 3: [...]
24        }
25       while (lines--) {
26         s->frame.data[0][pixel_ptr + 0] =
27           s->codebook[vector_index++];
28         s->frame.data[0][pixel_ptr + 1] =
29           s->codebook[vector_index++];
30         s->frame.data[0][pixel_ptr + 2] =
31           s->codebook[vector_index++];
32         s->frame.data[0][pixel_ptr + 3] =
33           s->codebook[vector_index++];
34         pixel_ptr += s->frame.linesize[0];
35         }
36     }
37   }
38 }
39
40 static av_cold int vqa_decode_init(AVCodecContext *avctx)
41 {
42     VqaContext *s = avctx->priv_data;
43     unsigned char *vqa_header;
44     int i, j, codebook_index;
45     s->avctx = avctx;
46     avctx->pix_fmt = PIX_FMT_PAL8; [...]
47       /* load up the VQA parameters from the header */
48     vqa_header = (unsigned char *)s->avctx->extradata;
49     s->vqa_version = vqa_header[0];
50      s->width = AV_RL16(&vqa_header[6]);
51     s->height = AV_RL16(&vqa_header[8]); [...]
52     /* allocate decode buffer */
53     s->decode_buffer_size = (s->width / s->vector_width) *
54         (s->height / s->vector_height) * 2;
55     s->decode_buffer = av_malloc(s->decode_buffer_size);
56     s->frame.data[0] = NULL;
57     return 0;
58 }
```

Figure 6: The second zero-day vulnerability found by extrapolation of CVE-2010-3429 in `vqa_decode_chunk`.

3. The function `vqa_decode_chunk` contains another previously unknown vulnerability shown in Figure 6. In this case, however, reading of frame meta data using the characteristic API is performed by a different function, `vqa_decode_init`, on line 21 and 22. Despite the missing API information, our method is able to uncover the decoder based on its characteristic code structure. In particular, the two nested loops iterating over the pixel buffer on line 6 and 8 are common to most decoders.

In summary, this example shows that our method is useful to identify zero-day vulnerabilities in real-world code. Furthermore, by combining information about symbol usage and code structure, our method gains increased robustness over approaches resting solely on symbol information.

Sim.	Function name	Sim.	Function name
0.98	flic_decode_frame_15_16BPP	0.87	wmavoice_decode_init
0.92	decode_frame	0.85	decode_frame
0.92	decode_frame	0.84	smc_decode_stream
0.91	flac_decode_frame	0.84	rl2_decode_init
0.90	decode_format80	0.84	xvid_encode_init
0.89	decode_frame	0.84	vmdvideo_decode_init
0.89	tgv_decode_frame	0.83	mjpega_dump_header
0.89	vmd_decode	0.82	ff_flac_is_._valid
0.89	wavpack_decode_frame	0.82	decode_init
0.88	adpcm_decode_frame	0.82	ws_snd_decode_frame
0.88	decode_frame	0.81	bmp_decode_frame
0.88	aasc_decode_frame	0.81	sbr_make_f_master
0.88	vqa_decode_chunk	0.80	ff_h264_decode_ref_pic_.
0.87	cmv_process_header	0.80	decode_frame
0.87	msrle_decode_8_16_24_32	0.79	vqa_decode_init

Table 2: Top 30 most similar functions to a known vulnerability in FFmpeg.

3.2.2 *Case study: Pidgin*

To demonstrate the generality of our method, we perform a second extrapolation on a different code base and for a different vulnerability type. In this case study, the set of relevant functions is very small in comparison to the size of the code base, that is, only 67 out of 11,505 functions are candidate functions. Despite this increased difficulty, we are able to identify nine vulnerabilities similar to a known vulnerability among the first 30 hits, six of which were previously unknown.

Original Vulnerability. The function `receiveauthgrant` shown in Figure 7 does not perform sufficient validation of UTF-8 strings received from the network allowing attackers to cause a denial of service condition and possibly execute arbitrary code. The function reads the username and message from an incoming binary data packet on line 15 and 20 respectively. It then passes these values to a suitable packet handler on line 27. In general, packet handlers assume that strings passed in parameters are valid UTF8-strings, however, the function does not ensure that this is the case.

Extrapolation. We again apply our method to obtain the 30 most similar functions to the original vulnerability as shown in Table 3. We first note that 28 of the first 30 hits are candidates selected from a total of 11,505 functions. After a short inspection of the suggested functions, we are able to spot the same type of vulnerability as in the original function in nine of the candidates. As an example, consider function `parseicon` shown in Figure 7. The username is read from the binary data packet on line 16 and is passed to a handler function unchecked on line 25–similar to the vulnerability in `receiveauthgrant`. It has been verified that this again allows to cause a denial of service condition.

The second case study demonstrates that our method works well even when the number of relevant functions is small compared to the total size of the code base. We are able to narrow down the search for potentially vulnerable

```
static int
receiveauthgrant(OscarData *od,
                 FlapConnection *conn,
                 aim_module_t *mod,
                 FlapFrame *frame,
                 aim_modsnac_t *snac,
                 ByteStream *bs)
{
  int ret = 0;
  aim_rxcallback_t userfunc;
  guint16 tmp;
  char *bn, *msg;
  /* Read buddy name */
  if ((tmp = byte_stream_get8(bs)))
    bn = byte_stream_getstr(bs, tmp);
  else
    bn = NULL;
  /* Read message (null terminated) */
  if ((tmp = byte_stream_get16(bs)))
    msg = byte_stream_getstr(bs, tmp);
  else
    msg = NULL;
  /* Unknown */
  tmp = byte_stream_get16(bs);
  if ((userfunc =
       aim_callhandler(od, snac->family, snac->subtype)))
    ret = userfunc(od, conn, frame, bn, msg);
  g_free(bn);
  g_free(msg);
  return ret;
}
```

```
static int
parseicon(OscarData *od,
          FlapConnection *conn,
          aim_module_t *mod,
          FlapFrame *frame,
          aim_modsnac_t *snac,
          ByteStream *bs)
{

  int ret = 0;
  aim_rxcallback_t userfunc;
  char *bn;
  guint16 flags, iconlen;
  guint8 iconcsumtype, iconcsumlen, *iconcsum, *icon;

  bn = byte_stream_getstr(bs, byte_stream_get8(bs));
  flags = byte_stream_get16(bs);
  iconcsumtype = byte_stream_get8(bs);
  iconcsumlen = byte_stream_get8(bs);
  iconcsum = byte_stream_getraw(bs, iconcsumlen);
  iconlen = byte_stream_get16(bs);
  icon = byte_stream_getraw(bs, iconlen);
  if ((userfunc =
       aim_callhandler(od, snac->family, snac->subtype)))
    ret = userfunc(od, conn, frame, bn, iconcsumtype,
                   iconcsum, iconcsumlen, icon, iconlen);
  g_free(bn);
  g_free(iconcsum);
  g_free(icon);
  return ret;
}
```

Figure 7: Original vulnerability (CVE-2011-4601) in `receiveauthgrant` (left), zero-day vulnerability in `parseicon` (right).

Sim.	Function name	Sim.	Function name
1.00	receiveauthgrant	0.98	incomingim_ch4
1.00	receiveauthreply	0.98	parse_flap_ch4
1.00	parsepopup	0.98	infoupdate
1.00	parseicon	0.98	parserights
1.00	generror	0.98	incomingim
0.99	incoming_..._buddylist	0.98	parseadd
0.99	motd	0.97	userinfo
0.99	receiveadded	0.97	parsemod
0.99	mtn_receive	0.97	parsedata
0.99	msgack	0.97	rights
0.99	keyparse	0.97	rights
0.99	hostversions	0.97	uploadack
0.98	userlistchange	0.96	incomingim_ch2_sendfile
0.98	migrate	0.96	rights
0.98	error	0.96	parseinfo_create

Table 3: Top 30 most similar functions to a known vulnerability in Pidgin.

code to a handful of functions. Again, the extraploation enables us to identify previously unknown vulnerabilities by inspecting only a small fraction of the code base.

4. LIMITATIONS

The discovery of vulnerable code in software is a hard problem. Due to the fundamental inability of one program to completely analyze another program's code, a generic technique for finding arbitrary vulnerabilities does not exist [10]. As a consequence, all practical approaches either limit the search to specific types of vulnerabilities or, as in the case of vulnerability extrapolation, only identify *potentially* vulnerable code. In the following we discuss the limitations of our approach in more detail.

Our method builds on techniques from machine learning, such as the embedding in a vector space and latent semantic analysis. These techniques are effective in identifying potentially vulnerable code, yet they do not provide any guarantees whether the identified code truly contains a vulnerability. This limitation is inherent to the application of machine learning, which considers the statistics of the source code rather than the true semantics. Due to Rice's theorem, however, a generic discovery of vulnerabilities is impossible anyway and thus even the discovery of potential vulnerabilities is beneficial in practice.

A prerequisite for extrapolation is the existence of a starting vulnerability. In cases where no known vulnerability is available, our method cannot be applied. In practice such cases are rare. For large software projects, it is not the discovery of a single vulnerability that is challenging but making sure that similar flaws are not spread across the entire code base. Extrapolation addresses exactly this setting. Moreover, related techniques such as fuzz testing, taint analysis and symbolic execution can be easily coupled with vulnerability extrapolation and provide starting vulnerabilities automatically.

Finally, the discovery of our method is limited to vulnerabilities present in few functions of source code. Complex flaws that span several functions across a code base can be difficult to detect for our method. However, our case study with FFmpeg shows that vulnerabilities distributed over two functions can still be effectively identified, as long as both functions share some structural patterns with the original vulnerability.

5. RELATED WORK

The identification of vulnerabilities has been a vivid area of security research. Various contrasting concepts have been devised for finding and eliminating security flaws in source

code. Our method is related to several of these approaches, as we point out in this section.

5.1 Code Clone Detection

In the simplest case, functions containing similar vulnerabilities exist because code has been copied. The detection of such *copy-&-paste code clones* has been an ongoing research topic. In particular, Kontogiannis et al. [14] explore the use of numerical features, such as the number of called functions, to detect code clones, while Baxter et al. [2] suggest a more fine-grained method, which compares ASTs. Li et al. present *CP-Miner*, a tool for code clone detection based on frequent itemset mining [16]. They demonstrate the superiority of their approach to the well-known tool *CCFinder* developed by Kamiya et al. [13], a token-based detection method. Maletic et al. [19] propose a method for code clone detection, which compares functions in terms of comments and identifiers. Similarly, Jang et al. have introduced a method for finding unpatched copies of code using n-gram analysis [11]. A thorough evaluation of existing methods is provided by Bellon et al. [3].

Code clone detection shares some similarities with vulnerability extrapolation. However, corresponding methods address a fundamentally different problem and are specifically tailored to finding copied code. As result, they can only uncover vulnerabilities that have been introduced by duplication of code.

5.2 Static Code Analysis

The idea of vulnerability extrapolation hinges on the observation that patterns of API usage are often indicative for vulnerable code. This correspondence has been recognized for a long time and is reflected in several static analysis tools, such as *Flawfinder* [30], *RATS* [24] or *ITS4* [28]. These tools offer databases of API symbols commonly found in conjunction with vulnerabilities and allow a code base to be automatically scanned for their occurrences. The effectiveness of these tools critically depends on the quality and coverage of the databases. Vulnerabilities related to internal APIs or unknown patterns of API usage cannot be uncovered.

Engler et al. [6] are among the first to explore the link between vulnerabilities and programming patterns. Their method is capable of detecting vulnerabilities given a set of manually defined programming patterns. As an extension, Li and Zhou [15] present an approach for mining similar patterns automatically and detecting their violation in code. An inherent problem of this approach is that frequent programming mistakes will lead to the inference of valid patterns and thus common flaws cannot be detected. Williams et al. [31] and Livshits et al. [17] address this problem by incorporating software revision histories into the analysis. Our method is related to these approaches. However, we focus on the analysis of code structure for finding vulnerabilities, rather than modelling common programming templates.

5.3 Taint Analysis and Symbolic Execution

A more generic approach to vulnerability discovery builds on taint analysis, where vulnerabilities are identified by a source-sink system. If tainted data stemming from a source propagates to a sink without undergoing validation, a potential vulnerability is detected. The success of this approach has been demonstrated for several types of vulnerabilities, including SQL injection and cross-site scripting [12, 18] as well as integer-based vulnerabilities [29]. In most realizations, taint analysis is a dynamic process and thus limited to discovery of vulnerabilities observable during execution of a program.

The limitations of taint analysis have been addressed by several authors using symbolic execution [e.g., 1, 4, 8, 22]. Instead of passively monitoring the flow from the a source to a sink, these approaches try to actively explore different execution paths. Most notably is the work of Avgerinos et al. [1] that introduces a framework for finding and even exploiting security vulnerabilities. The power of symbolic execution, however, comes at a prize: the analysis often suffers from a vast space of possible execution paths. In practice, different assumptions and heuristics are necessary to trim down this space to a tractable number of branches. As a consequence, the application of symbolic execution for regular code auditing is still far from being practical [9].

5.4 Vulnerability Extrapolation

The concept of vulnerability extrapolation has been first introduced in [33]. In this work a method for vulnerability extrapolation is proposed that analyzes the usage of function and type names for finding vulnerabilities. However, neither the syntax nor the structure of the code are considered and thus the analysis is limited to flaws reflected in particular API symbols. Our method significantly extends this work by extracting and analyzing structural patterns in ASTs. As a result, we are able to discover vulnerabilities that are related to API usage as well as the structure of code. Moreover, we demonstrate the efficacy of our approach on a significantly larger set of code.

In comparison with related approaches, it is noteworthy that vulnerability extrapolation does not aim at identifying vulnerabilities automatically, but rendering manual auditing of source code more effective. The underlying rationale is that manual auditing—though time-consuming—is still superior to automatic methods and indispensable in practice. The assisted approach of vulnerability extrapolation here better fits the needs of security practitioners.

6. CONCLUSIONS

A key to strengthening the security of computer systems is the rigorous elimination of vulnerabilities in the underlying source code. To this end, we have introduced a method for accelerating the process of manual code auditing by suggesting potentially vulnerable functions to an analyst. Our method extrapolates known vulnerabilities using structural patterns of the code and enables efficiently finding similar flaws in large code bases. Empirically, we have demonstrated this capability by identifying real zero-day vulnerabilities in open-source projects, including Pidgin and FFmpeg.

The concept of vulnerability extrapolation is orthogonal to other approaches for finding vulnerabilities and can be directly applied to complement current instruments for code auditing. For example, if a novel vulnerability is identified using fuzz testing or symbolic execution, it can be extrapolated to the entire code base, such that similar flaws can be immediately patched. This extrapolation raises the bar for attackers, as they are required to continue searching for novel vulnerabilities, once an existing flaw has been sufficiently extrapolated and related holes have been closed in the source code.

Reporting of Vulnerabilities

The discovered vulnerabilities have been reported to the respective developers before submission of this paper. The flaws should be fixed in upcoming versions of the projects.

References

[1] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. AEG: Automatic Exploit Generation. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2011.

[2] I. D. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proc. of the International Conference on Software Maintenance (ICSM)*, 1998.

[3] S. Bellon, R. Koschke, I. C. Society, G. Antoniol, J. Krinke, I. C. Society, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33:577–591, 2007.

[4] M. Cova, V. Felmetsger, G. Banks, and G. Vigna. Static detection of vulnerabilities in x86 executables. In *Proc. of Annual Computer Security Applications Conference (ACSAC)*, pages 269–278, 2006.

[5] S. Deerwester, S. Dumais, G. Furnas, T. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407, 1990.

[6] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proc. of ACM Symposium on Operating Systems Principles (SOSP)*, pages 57–72, 2001.

[7] N. Falliere, L. O. Murchu, , and E. Chien. W32.stuxnet dossier. Symantec Corporation, 2011.

[8] P. Godefroid, M. Y. Levin, and D. Molnar. SAGE: whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012.

[9] S. Heelan. Vulnerability detection systems: Think cyborg, not robot. *IEEE Security & Privacy*, 9(3):74–77, 2011.

[10] J. Hopcroft and J. Motwani, R. Ullmann. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2 edition, 2001.

[11] J. Jang, A. Agrawal, , and D. Brumley. ReDeBug: finding unpatched code clones in entire os distributions. In *Proc. of IEEE Symposium on Security and Privacy*, 2012.

[12] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *Proc. of IEEE Symposium on Security and Privacy*, pages 6–263, 2006.

[13] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, pages 654–670, 2002.

[14] K. A. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *Journal of Automated Software Engineering*, 3:108, 1996.

[15] Z. Li and Y. Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proc. of European Software Engineering Conference (ESEC)*, pages 306–315, 2005.

[16] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32:176–192, 2006.

[17] B. Livshits and T. Zimmermann. Dynamine: finding common error patterns by mining software revision histories. In *Proc. of European Software Engineering Conference (ESEC)*, pages 296–305, 2005.

[18] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proc. of USENIX Security Symposium*, 2005.

[19] A. Marcus and J. I. Maletic. Identification of high-level concept clones in source code. In *Proc. of International Conference on Automated Software Engineering (ASE)*, page 107, 2001.

[20] L. Moonen. Generating robust parsers using island grammars. In *Proc. of Working Conference on Reverse Engineering (WCRE)*, pages 13–22, 2001.

[21] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the Slammer worm. *IEEE Security and Privacy*, 1(4):33–39, 2003.

[22] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2005.

[23] T. Parr and R. Quong. ANTLR: A predicated-LL(k) parser generator. *Software Practice and Experience*, 25: 789–810, 1995.

[24] rats. Rough auditing tool for security. Fortify Software Inc., `https://www.fortify.com/ssa-elements/threat-intelligence/rats.html`, visited April, 2012.

[25] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, 1986.

[26] C. Shannon and D. Moore. The spread of the Witty worm. *IEEE Security and Privacy*, 2(4):46–50, 2004.

[27] M. Sutton, A. Greene, and P. Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.

[28] J. Viega, J. Bloch, Y. Kohno, and G. McGraw. ITS4: A static vulnerability scanner for C and C++ code. In *Proc. of Annual Computer Security Applications Conference (ACSAC)*, pages 257–267, 2000.

[29] T. Wang, T. Wei, Z. Lin, and W. Zou. IntScope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2009.

[30] D. A. Wheeler. Flawfinder. `http://www.dwheeler.com/flawfinder/`, visited April, 2012.

[31] C. C. Williams and J. K. Hollingsworth. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Transactions on Software Engineering*, 31:466–480, 2005.

[32] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proc. of USENIX Security Symposium*, 2006.

[33] F. Yamaguchi, F. Lindner, and K. Rieck. Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning. In *USENIX Workshop on Offensive Technologies (WOOT)*, Aug. 2011.

XIAO: Tuning Code Clones at Hands of Engineers in Practice

Yingnong Dang[1], Dongmei Zhang[1], Song Ge[1], Chengyun Chu[2], Yingjun Qiu[3*], Tao Xie[4]
[1]Microsoft Research Asia, China
[2]Microsoft Corporation, USA
[3]Alibaba Corporation, China, [4]NC State University, USA
{yidang;dongmeiz;songge;chchu}@microsoft.com, soloqyj@msn.com, xie@csc.ncsu.edu

ABSTRACT
During software development, engineers often reuse a code fragment via copy-and-paste with or without modifications or adaptations. Such practices lead to a number of the same or similar code fragments spreading within one or many large codebases. Detecting code clones has been shown to be useful towards security such as detection of similar security bugs and, more generally, quality improvement such as refactoring of code clones. A large number of academic research projects have been carried out on empirical studies or tool supports for detecting code clones. In this paper, we report our experiences of carrying out successful technology transfer of our new approach of code-clone detection, called XIAO. XIAO has been integrated into Microsoft Visual Studio 2012, to be benefiting a huge number of developers in industry. The main success factors of XIAO include its high tunability, scalability, compatibility, and explorability. Based on substantial industrial experiences, we present the XIAO approach with emphasis on these success factors of XIAO. We also present empirical results on applying XIAO on real scenarios within Microsoft for the tasks of security-bug detection and refactoring.

Categories and Subject Descriptors
D.2.7 [Software Engineering]: [Distribution, Maintenance, Enhancement]

General Terms
Security, Algorithm

Keywords
Code clone, code duplication, duplicated security vulnerability, code-clone detection, code-clone search

1. INTRODUCTION
During software development, engineers often reuse a code fragment via copy-and-paste with or without modifications or adaptations. Such practices lead to a number of the same or similar code fragments called code clones spreading within one or many large codebases. Detecting code clones [6][10][14][18][20] has been commonly shown to be useful towards various software-engineering tasks such as bug detection and refactoring.

In general, there are four main types of code clones [6][20]. Type-I clones are identical code fragments except for variations in whitespace, layout, or comments. Type-II clones are syntactically identical fragments except for variations in identifiers, literals, types, whitespace, layout, or comments. Type-III clones are copied fragments with further modifications such as changed, added, or removed statements, in addition to variations in identifiers, literals, types, whitespace, layout, or comments. Type-IV clones are code fragments that perform similar functionality but are implemented by different syntactic variants.

Among these four types of code clones, type-III code clones with or without disordered statements, called *near-miss code clones*, are of high practical interest because they may potentially have a negative impact on the code quality and increase maintenance cost [10]. For example, problems might occur when some code is changed for fixing a bug but the same fix is not applied to its clones. Another example is inconsistent evolution of code clones, e.g., one piece of code is changed for supporting more data types, but its clones are not changed accordingly. Figure 1 shows an example near-miss clone (which indicates a bug) reported by a Microsoft engineer. The difference between the code snippets A and B is relatively large: one statement in the code snippet B (Line 16) is replaced by 4 statements in code snippet A (Lines 16-19), and the "if" statement in code snippet B (Lines 23-25) is updated as Lines 24-28 in A with significant changes in the "if" condition.

A large number of academic research projects [20] have been carried out on empirical studies or tool supports for detecting code clones. However, in practice, so far few such research projects have resulted in substantial industry adoption beyond the empirical studies conducted by researchers themselves. Although a few integrated development environments have integrated the generic feature of code-clone detection, this feature has limited support for real use in practice, and no industrial experiences are reported on the application of such feature.

In this paper, we attempt to address this issue and share to the community with experiences of carrying out successful technology transfer of our new approach of code-clone detection [8], called XIAO. XIAO has already been used by a large number of Microsoft engineers in their routine development work, especially engineers from a security-engineering team at Microsoft who have been using XIAO's online clone-search service since May 2009 to help with their investigation on security bugs. XIAO has been integrated into Microsoft Visual Studio 2012, to be benefiting a huge number of engineers in industry.

Based on our experiences [8] of collaborating with Microsoft engineers on using and improving XIAO along with our

---

* This work was done when this author worked for Microsoft Research Asia.

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

```
    // 3 identical statements omitted here
4.  switch (biBitCount)
5.  {
    // 9 identical statements omitted here
15. case 24: // 24bpp: Read colours from pixel
16. case 32:
17. palEntry.rgbRed    = ((RGBQUAD *)pPixel)->rgbRed;
18. palEntry.rgbGreen = ((RGBQUAD *)pPixel)->rgbGreen;
19. palEntry.rgbBlue    = ((RGBQUAD *)pPixel)->rgbBlue;
20. break;
21. default: // What else could it be?
22. return 0;
23. }
24. if (palEntry.rgbRed >= 0xFE && palEntry.rgbGreen >= 0xFE &&
25. palEntry.rgbBlue >= 0xFE ||((palEntry.rgbRed >= 0xbf &&
26. palEntry.rgbGreen >= 0xbf && palEntry.rgbBlue >= 0xbf) &&
27. (palEntry.rgbRed <= 0xc1 && palEntry.rgbGreen <= 0xc1 &&
28. palEntry.rgbBlue <= 0xc1)))
29.     return FALSE;
30. return TRUE;
```

Code Snippet A

```
    // 3 identical statements omitted here
4.  switch (biBitCount)
5.  {
    // 9 identical statements omitted here
15. case 24: // 24bpp: Read colours from pixel
16.     palEntry = *(RGBQUAD *)pPixel;
17.       break;
18.
19.
20.      default:    // What else could it be?
21.      return 0;
22.   }
23.   if (palEntry.rgbRed == 0xFF && palEntry.rgbGreen ==
    0xFF
24.     &&palEntry.rgbBlue == 0xFF || palEntry.rgbRed == 0xC0
25.     palEntry.rgbGreen == 0xC0 && palEntry.rgbBlue == 0xC0)
26.
27.       return FALSE;
28.    return TRUE;
```

Code Snippet B

Figure 1. An example of near-miss code clones in a commercial codebase

observations on real use of XIAO by Microsoft engineers, we attribute the success of XIAO to four main factors: its high tunability, scalability, compatibility, and explorability.

High tunability of XIAO is achieved with a new set of similarity metrics in XIAO, reflecting What You Tune Is What You Get (WYTIWYG): users can intuitively relate tool-parameter values with the tool outputs, and easily tune tool-parameter values to produce what the users want. For example, the similarity-parameter value of 100% should lead to outputs of two exactly same cloned snippets, and the 80% value should lead to outputs of two cloned snippets with 80% similarity judged by the users. The parameters of the proposed metrics in XIAO enable users to effectively control the degree of the syntactic difference between the two code snippets of a near-miss clone pair: the degree of the statement similarity, the percentage of inserted/deleted/modified statements in the clone pair, the balance between the code-structure similarity, and the quantity of disordered statements. Such high tunability of XIAO is critical in applying an approach of code-clone detection such as XIAO to a broad scope of software-engineering tasks such as refactoring and bug detection since these different tasks would require different levels of parameter values.

High scalability of XIAO in analyzing enormous lines of code is achieved with a well-designed scalable and parallelizable algorithm with four steps. These four steps include preprocessing, coarse matching, fine matching, and pruning. Preprocessing transforms source-code information to filter out inessential information such as code comments, and map code entities such as keywords and identifiers to tokens. Such information preprocessing reduces the cost burden of the actual analysis. To offer high scalability, XIAO splits the main analysis into two steps: coarse matching and fine matching. Coarse matching is less costly but less accurate than fine matching. The scope narrowed down by coarse matching is fed to fine matching, achieving a good balance on analysis scalability and accuracy. The step of pruning further improves the analysis accuracy. In addition, the clone-detection algorithm of XIAO can be easily parallelized. XIAO partitions the codebase and performs code-clone detection on each code-partition pair. Each instance of XIAO detects clones on a number of pairs. The results of all the instances are then merged.

High compatibility of XIAO in analyzing code in different development environments (such as different build systems) is achieved with its compiler-independent lightweight and pluggable parsers. XIAO has built-in parsers for the C/C++ and C# languages. We define an open Application Programming Interface that allows the easy plug-in of parsers to support various programming languages. It should be noted that the parsing task is lighter than the comprehensive functionalities offered by compilers. Compared with approaches of parse-tree-based clone detection such as Deckard [14][9], our approach has the advantage of compiler independence; it can be easily applied to accommodate different language variants and build environments, which typically exist in real settings of software development, especially for C/C++ [7].

High explorability of XIAO in supporting users to easily explore and manipulate detected code clones is achieved with its well-designed user interfaces including visualization support. We design a simple heuristic to define the level of difference between cloned snippets. We also use the metric to rank clones to prioritize the review of clones to identify bugs. XIAO includes clone visualization to clearly show the matching blocks and the block types of a clone pair. This way, users can quickly capture whether there is any difference between the two cloned snippets, what kind of difference it is, and how much difference there is. XIAO also includes a tagging mechanism to help coordinate joint efforts of reviewing code clones from multiple engineers.

We have released XIAO to Microsoft engineers since April 2009 and a great number of Microsoft engineers from different teams have used it. XIAO has been integrated into Microsoft Visual Studio 2012, to be benefiting a huge number of engineers in industry.

The rest of this paper is organized as follows. We present our code-similarity metric in Section 2. We introduce our clone-detection algorithm and visualization/reporting in Section 3 and Section 4, respectively. We present our empirical study in Section 5 and report several real-use scenarios in Section 6. Section 7 discusses related work and Section 8 concludes.

2. CODE SIMILARITY METRIC

Considering possible edits that can be applied to source code after it has been copied and pasted, we have identified three important

inconsistencies that are present in near-miss clones and that should be measured:

1. Statement-level difference, which may be caused by adapting the copied code to the coding style of the source file where it is copied to, e.g., a few identifiers are renamed;
2. Inserted/deleted/modified statements, which can be the result of changes to the code necessary for fixing bugs or implementing new features;
3. Disordered statements, which may be related to either a change of code logic or coding style.

We define our code-similarity metric to take these three types of inconsistencies into account. Before we define our metric, we next define three binary relationships between source-code statements. Let $\sum$ be an alphabet whose symbols are syntactical tokens, and $s \in \sum^*$ be a source-code statement.

Definition 1 (Exact-Match) Let s_1 and s_2 be two source-code statements; let $|s|$ be the token length of the statement *s*, and *s[i]* be the *i*-th token of *s*. Then s_1 and s_2 are *Exact-Match*-related if and only if $|s_1| = |s_2|, and\ s_1[i] = s_2[i],\ 0 < i \leq |s_1|$.

Definition 2 (Transformed-Match) Let $T: \sum \to \sum$ be a token-mapping function (e.g., mapping several different identifiers to the same token). The statements s_1 and s_2 are *Transformed-Match*-related if and only if $|s_1| = |s_2|, and\ T(s_1[i]) = T(s_2[i]),\ 0 < i \leq |s_1|$.

Definition 3 (α-Transformed-Match) The statements s_1 and s_2 are *α-Transformed-Match*-related if and only if they are *Transformed-Match*-related and there exist at least $\alpha * |s_1|$ (denoted as l) distinct indexes $i_1, \ldots, i_l$ such that $s_1[i_k] = s_2[i_k], 0 < k \leq l$.

Intuitively, two statements are *Exact-Match*-related if they are identical after code formatting is ignored, and are *Transformed-Match*-related if they are identical after both code formatting and identifier renaming are ignored. The *α-Transformed-Match* relationship provides the flexibility for controlling the degree on what percentage of renamed identifiers between two matched statements are tolerated. In fact, the *α-Transformed-Match* relationship is equal to the *Transformed-Match* relationship when $\alpha = 0$, and equal to the *Exact-Match* relationship when $\alpha = 1$.

We next illustrate these three relationships with examples. Consider that the parameterized versions of three statements A, B, and C listed below are identical.

Original statements	Parameterized statements
A: If (foo(a, b, c) == null)	A1: If (T (T, T, T) == T)
B: If (foo(a, b1,c1) == null)	B1: If (T (T, T, T) == T)
C: If (bar(x, y, z) == null)	C1: If (T (T, T, T) == T)

Consequently, any two of them are *Transformed-Match*-related. Each statement has 13 tokens. Statements A and B have 11 identical tokens out of the 13 tokens (84.6%); Statements A and C have 9 identical tokens out of the 13 tokens (69.2%). If $\alpha = 0.6$, A is *α-Transformed-Match*-related to both B and C; if $\alpha = 0.8$, A is *α-Transformed-Match*-related to only B.

In practice, Statements A and C could have a totally different semantic and could not be caused by copy-and-paste, thus being of low interest to users who focus on copy-and-paste clones.

Definition 4 (Disordered-Match-Score, in short as DMS) Consider two code snippets S_1 and S_2 where *m* distinct statements in S_1 at positions $i_1 < i_2 < \cdots < i_m$ match *m* distinct statements at positions $i'_1, i'_2, \cdots, i'_m$ in S_2. Let *d* be the number of inversion pairs[1] in the index sequence $\{i'_1, i'_2 \cdots i'_m\}$. The *Disordered-Match-Score (DMS)* of S_1 and S_2 is defined as

$$DMS(S_1, S_2) = \frac{2*d}{m-1} \quad (1)$$

The *DMS* measures the structure difference of the two code snippets. The smaller the value of the *DMS* is, the more similar the structures of the two code snippets are because there are fewer disordered statements. The possible value of the *DMS* is from 0 (when there is no inversion pair) to *m* (when the order is fully reversed). Figure 2 shows an example of two code snippets with disordered statements: the statement in Line 4 on the left snippet is moved to Line 2 on the right. The statements in Lines 1 to 5 on the left snippet correspond to the statements in Lines 1, 3, 4, 2, and 5 on the right snippet. The inversion pairs of this index sequence are (3, 2) and (4, 2). The *DMS* of the two snippets in the example is 1 (being 2*2/4) .

```
1  for (i = 0; i < 10; i++) {        1  for (i = 0; i < 10; i++) {
2     a++;                           2     c=foo(a, b);
3     b++;                           3     a++;
4     c=foo(a, b);                   4     b++;
5     d=bar(a, b, c);}               5     d=bar(a, b, c);}
```

Figure 2. Example of disordered statements

Definition 5 (α-Transformed Similarity) Let $|S|$ be the number of statements of a code snippet *S*. The *α-Transformed Similarity* between two snippets S_1 and S_2 is defined as

$$\alpha Sim(S_1, S_2) = 2\frac{m - \theta DMS(S_1, S_2)}{|S_1| + |S_2|} \quad (2)$$

where m is as in Definition 4 and θ is a penalty coefficient to the DMS. We use the α-Transformed similarity to measure code similarity hereafter.

Definition 6 (Transformed Similarity) The *Transformed Similarity*, $TSim(S_1, S_2)$ is defined as its *α-Transformed Similarity* with $\alpha = \theta = 0$.

Definition 7 (Clone Pair) Code snippets S_1 and S_2 are a clone pair when $\alpha Sim(S_1, S_2) \geq \gamma, 1 \geq \gamma > 0$, where γ is a configurable similarity threshold.

Definition 7 satisfies the requirement posed at the beginning of this section. It controls the statement-level difference by the value of α. It controls the number of inserted/deleted statements proportionally to the size of the code snippets by the value of γ; the bigger the snippets, the more inserted/deleted statements are tolerated. The penalty coefficient θ permits users to control the amount of disordered statements in a clone pair.

3. CLONE-DETECTION ALGORITHM

Figure 3 shows an overview of our clone-detection algorithm, consisting of four steps: preprocessing, coarse matching, fine matching, and pruning. Our algorithm takes one codebase as input and produces code clones detected from the codebase (our

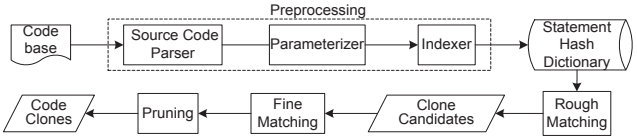


Figure 3. Overview of XIAO's algorithm of code-clone detection

[1] http://en.wiktionary.org/wiki/inversion_pair

algorithm can handle multiple codebases by treating them as one codebase).

In the preprocessing step, the source-code parser extracts the location information of all the functions and their statements. Then the code is parameterized and indexed similar to the preprocessing techniques of CP-Miner [21].

In the coarse-matching step, for each function f in the codebase, a list of its clone-candidate functions $\{\mathrm{CF_f}\}$ is detected. Each candidate function has a sufficient number of statements with the same hash value as at least one statement in f. This step helps reduce the search space of the fine-matching step.

In the fine-matching step, we identify all clone pairs between each function f and each of its clone-candidate functions $\{\mathrm{CF_f}\}$ using the metric in Definition 6, with $\alpha = \theta = 0$. The setting of $\alpha = 0$ enables us to use the hash values of the parameterized statements to easily verify the matching relationship; $\theta = 0$ enables us to easily calculate the similarity using Equation (2).

In the pruning step, we recalculate the similarity of the clone pairs obtained in the fine-matching step, using the user-specified non-zero values of α and θ, and thus prune the clone pairs with a similarity that is less than the similarity threshold γ. We next give the details of the last three steps in this section.

3.1 Coarse Matching

Given a function f and a statement-hash dictionary D, the coarse-matching algorithm returns a list of candidate functions $\{\mathrm{CF_f}\}$ so that at least a minimum and sufficient number of statements in f and any function in $\{\mathrm{CF_f}\}$ have the same hash values. Doing so ensures that only functions sharing a minimum and sufficient number of statements are searched for code clones. In this way, the search space is reduced from the whole input codebase to just $\{\mathrm{CF_f}\}$. All possible function pairs that potentially contain cloned code snippets are identified by performing the coarse matching on all the functions in the input codebase. The steps of fine matching and pruning are then performed between f and each function in $\{\mathrm{CF_f}\}$ to obtain actual clones.

We next define the concepts of the Hit Function and Clone Candidate Function to help illustrate the coarse-matching algorithm.

Definition 8 (Hit Function) Let $H: \sum^* \to \mathbb{N}$ be a hash function, and $T^*: \sum^* \to \sum^*$ be the extension of the token-mapping function T (see Definition 2) to whole statements. A function F_{hit} is named as a Hit Function of a function f if there exist a statement s in f and a statement s_h in F_{hit} that satisfy $H(T^*(s))) = H(T^*(s_h)))$.

Definition 9 (Clone-Candidate Function) A function CF_f is a Clone-Candidate Function of a function f if there exist at least n_{match} statements in f with CF_f as one of its Hit Functions and

$$n_{match} \geq \min\left(\frac{\gamma}{2-\gamma} \cdot L, \quad \gamma \cdot MinS\right) \qquad (3)$$

where L is the number of the statements in f, γ is the clone-similarity threshold in Definition 7, and $MinS$ is the minimal number of statements that a cloned snippet should have.

Intuitively, a Hit Function F_{hit} has at least one parameterized statement in common with function f. CF_f has at least n_{match} common parameterized statements with f.

Suppose that D is the hash dictionary of an input codebase. For every statement s in function f, the coarse-matching algorithm uses D to generate a list of Hit Functions $\{F_{hit}\}$ by retrieving functions each of which contains a parameterized statement with the same hash value as that of the parameterized form of s. $\uplus Hit(f)$ is the multiset union of all the functions in the Hit Function lists for every statement of f. The total hit count of each function in $\uplus Hit(f)$ is equal to the function's multiplicity in $\uplus Hit(f)$. We then identify a list of Clone-Candidate Functions of f as those functions in $\uplus \mathrm{Hit}(f)$ with no less than n_{match} occurrences.

3.2 Fine Matching

The coarse matching identifies a list of Clone-Candidate Functions for each function f in the input codebase. There may not be clone pairs between f and CF_f for the following reasons:

(a) the matched parameterized statements may be so scattered in f and CF_f that the similarity between the snippets in f and CF_f is not high enough;
(b) multiple parameterized statements in f or CF_f may be mapped to the same tokenized statement in CF_f or f, causing that the number of one-to-one matched statements between f and CF_f is not high enough;
(c) two statements are not necessarily α-*Transformed-Match*-related even if they have the same hash value;
(d) there might be mismatched statements between f and CF_f due to hash collisions, although the probability of hash collision is quite low;
(e) some matched statements could be instances of disordered matches and the penalty to the disordered match in Equation (2) would cause that the similarity is not high enough.

We address issues (a) and (b) in the fine-matching step and issues (c), (d), and (e) in the pruning step.

The goal of the fine-matching step is to identify all snippet pairs (between f and CF_f) whose Transformed Similarity (Definition 6) is not less than a specified threshold. We formulate this problem as finding code snippets S_1 and S_2 in f and CF_f, respectively, that satisfy

$$\left.\begin{array}{ll} H\left(T^*(s_{1,i})\right) = H\left(T^*(s_{2,i})\right) & (a) \\ \frac{2m}{|S_1|+|S_2|} \geq \gamma & (b) \end{array}\right\} \qquad (4)$$

where $s_{1,i}$ and $s_{2,i}$, (i = 1, … , m) are m statements in S_1 and S_2, respectively. Equation (4.a) ensures that there are m matching parameterized statements; Equation (4.b) ensures that the α-*Transformed Similarity* of S_1 and S_2 is not less than the similarity threshold γ given the values of α and θ in Equation (2) are equal to 0.

We next first present how to determine whether a given snippet pair S_1 and S_2 satisfies equation (4), and then present how to efficiently scan f and CF_f to find all the possible pairs of S_1 and S_2 in f and CF_f.

To determine whether S_1 and S_2 satisfy Equation (4), we calculate the value of m as follows. Suppose that (1) $\{V_i | i = 1,2,\cdots,t\}$ is the list of the hash values for which at least one statement in S_1 and one statement in S_2 are mapped to V_i, and (2) there are also $n_{1,i}$ and $n_{2,i}$ statements with the hash value V_i in S_1 and S_2, respectively. It easily follows that there are $n_i = \min(n_{1,i}, n_{2,i})$ matched parameterized statements in S_1 and S_2. Therefore, m can be easily calculated as

$$m = \sum_{i=1}^{t} n_i \qquad (5)$$

Accordingly, we determine whether S_1 and S_2 satisfy Equation (4).

The next subtask is to scan all the possible snippet pairs in f and CF_f. We take a two-step procedure. First, given a snippet S_1 in f,

we scan all the possible S_2 in CF_f and determine whether S_1 and S_2 satisfy Equation (4). Second, we enumerate all the possible S_1 in *f* and repeat the first step.

During the first step, we use a sliding window on top of the statement sequence of CF_f to enumerate all the code snippets in CF_f. The statement sequence inside the window is the current code snippet S_2. To satisfy Equation (4), the number of statements of S_2 in CF_f should satisfy the following constraint:

$$k_{min} \leq |S_2| \leq k_{max} \quad (6)$$

where $k_{min} = \frac{\gamma}{2-\gamma}|S_1|$, $k_{max} = \frac{2-\gamma}{\gamma}|S_1|$. Therefore, we need to use a set of sliding windows with sizes ranging from k_{max} to k_{min} to enumerate all possible snippets in CF_f. Given a sliding window size *k*, the window starts from position 1 (W_1) that covers the first *k* statements in CF_f. After checking whether the snippet inside the window and S_1 satisfy Equation (4), the window moves one step further to position 2 (W_2), and so on. Compared with the code snippet covered by W_1, the code snippet covered by W_2 has only the first statement of CF_f removed and the statement in position *k*+1 added. Therefore, we calculate the value of *m* for the code snippet in W_2 by just updating the value of *m* for the code snippet in W_1, i.e., by removing the contribution of the first statement and adding the contribution of the added statement in Equation (5).

During the second step, we use a sliding window to enumerate all the possible snippets S_1 in *f*, and repeat the first step. The size of this sliding window ranges from $|CF_f|$ (the total number of statements in CF_f) to *MinS* (the minimal number of statements that a cloned snippet should have).

We further optimize the algorithm in a number of ways. For example, the sliding windows in the first step could directly move to the next statement that matches at least one statement in *f*. In addition, once a snippet pair is identified as passing the fine matching, we further execute the pruning step against the pair to determine whether it is an actual clone pair or not. Once a snippet pair passes the pruning, we continue to perform the fine matching in the remaining parts of *f* and CF_f; in this way we avoid getting overlapped clone pairs.

3.3 Pruning

In the pruning step, we prune the snippet pairs obtained in the fine-matching step to get code clones that satisfy our code-clone definition with the specified non-zero values for α and θ in Equation (2). This step addresses issues (c), (d) and (e) mentioned at the beginning of Section 3.2.

To address these three issues, we need to get the *α-Transformed-Match*-related statements (Definition 2) in the two code snippets in the pair such that the *Disordered-Match-Score* (DMS) (Definition 4) of the two snippets is minimized. We then calculate the *α-Transformed-Similarity* based on Equation (2) and discard the snippet pair if its *α-Transformed-Similarity* value is lower than the threshold.

We use a greedy technique called Karp-Rabin Matching and Greedy String Tiling [30] to get the matched statements. The basic idea is to use a dynamic-programming algorithm to find the maximal consecutive statement sub-sequences $S^{1,1}$ in S_1, and $S^{2,1}$ in S_2, with the same number of statements, and each statement in $S^{1,1}$ *α-Transformed-Match*-related with the statement at the corresponding position in $S^{2,1}$. The next step is to exclude the statements in $S^{1,1}$ and $S^{2,1}$ from S_1 and S_2, respectively, and repeat the step on $S_1 \backslash S^{1,1}$ and $S_2 \backslash S^{2,1}$. By reiterating this process until there are no further matches, we get a set of statement-sub-sequence pairs in S_1 and S_2, which are *α-Transformed-Match-related* to each other. The matched statements that we need to obtain are the union of all the sub-sequence pairs. At this point, we calculate the *α-Transformed Similarity* and determine whether S_1 and S_2 are a clone pair based on Definition 7.

4. VISUALIZATION AND REPORTING

As important and integral components of XIAO, the clone visualization and reporting mechanism provides a rich and interactive user experience for engineers to efficiently review the clone-analysis results and take corresponding actions.

Clone reporting. We design a simple heuristic to define the level of difference between cloned snippets. In particular, it first filters out all those exactly the same cloned snippets, since cloned snippets with slightly different logics would be more bug-prone. We use a metric (called bug likelihood) to rank clones to prioritize the review of clones to identify bugs. We also design a simple heuristic to measure in what extent the cloned snippets are similar to each other and how easily they can be refactored (e.g., the exact same copies could be easier to be refactored than others). We call this metric as refactoring likelihood. To facilitate users to act on the reported clones, we have developed XIAO's Clone Explorer, a component of clone reporting and exploration shown in Figure 4. It organizes clone statistics based on the directory hierarchy of source files in order to enable quick and easy review at different source levels (Figure 4①). A drop-down list (②) is provided to allow pivoting the clone-analysis results around the bug likelihood (③), refactoring likelihood, and clone scope. Clone scope indicates whether cloned snippets are detected inside a file, cross-file, or cross-folder. For a selected folder in the left pane, the right pane (④) displays the list of clone functions (those including cloned snippets), which could be sorted based on bug likelihood or refactoring likelihood (⑥). Filters (⑤) on the clone scope, bug likelihood, or refactoring likelihood are provided to enable easy selection of clones of interest.

Clone visualization. Figure 5 shows how the Clone-Visualizer component visualizes the clone pair illustrated in Figure 1. The key to clone visualization is to clearly show the matched statement blocks and the block types. We categorize the matched blocks into the following types: exactly same (i.e., there are only possible formatting differences), similar-logic block (i.e., there are identifier substitutions between the two blocks), different logic (i.e., the statements in the two blocks are not of the similar-logic type but are still similar), and extra logic (i.e., the statements of a block show up in one copy of the clone pair, but not in the other copy). In this way, users can quickly determine whether there is any difference between the two cloned snippets, what kind of difference it is, and how much difference there is. Blocks are numbered for correspondence display (Figure 5 ①), and different colorings are used to indicate different block types (②). The left and right source panes are synchronized, and navigation buttons are provided to navigate through source code by matched blocks instead of statements in order to improve review efficiency (③). Users can take an immediate action of filing a bug once a clone is confirmed to be a bug or a refactoring target (⑤), or copying the code out for more investigation (④).

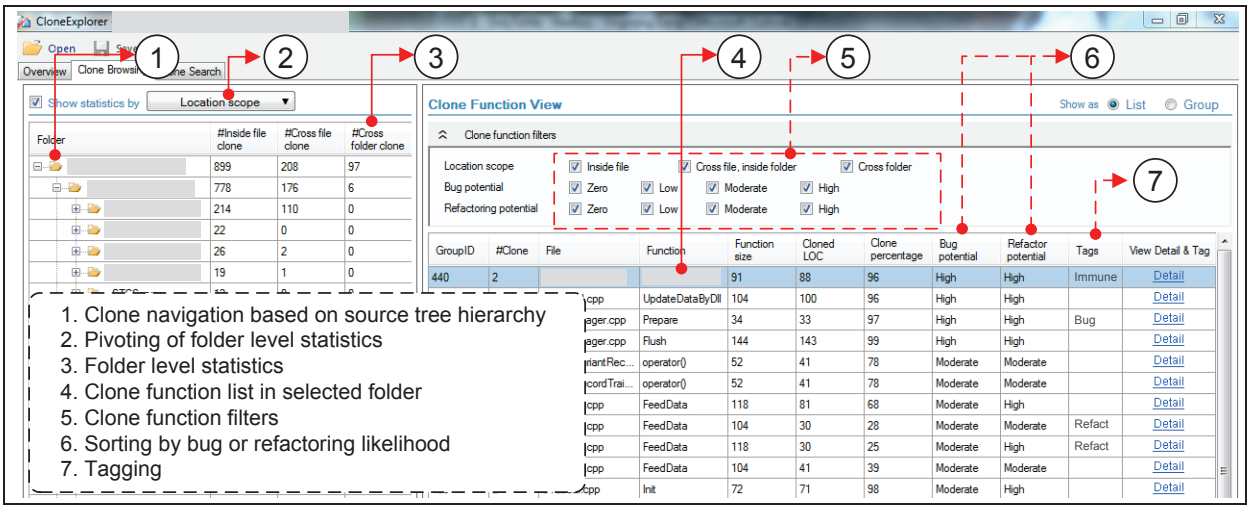


Figure 4. UI of Clone Explorer

Figure 5. Visualizing differences of a clone pair

Tagging. One important requirement of XIAO is to help coordinate joint efforts of reviewing code clones from multiple engineers. We have designed a tagging mechanism for engineers to easily work together. One clone already reviewed by an engineer can be tagged as "immune"[2], "bug", or "refactoring". Then the other reviewing engineers could choose to easily skip these already reviewed clones. Note that these tags need to be tracked as done in XIAO when a new version of codebase is analyzed. Overall, a tagging mechanism (Figure 4 ⑦) serves two main purposes. First, users can tag some clones as "immune" at various occasions. For example, some detected clones do not include buggy code or become refactoring targets. Second, we can implicitly collect user feedback and evaluation results in order to keep improving our clone-analysis algorithms.

5. EMPIRICAL STUDIES

In this section, we present the empirical results of applying XIAO on commercial codebases. In our studies, we used seven commercial codebases at Microsoft. In the seven commercial codebases, six are in C/C++ and one is in C#; the numbers of lines of code vary between 1.9 million and 12 millions.

The environment for running XIAO was a workstation running Windows 7 64 bits with two Intel Xeon 2.0GHz processors and 12GB memory. We relied on human inspection to classify whether a detected clone is a real clone.

5.1 Clone-Detection Effectiveness

Figure 6 shows the distribution of the types of code clones detected by XIAO across the seven commercial codebases, when using the default settings: *MinS* = 10, α = 0.6, γ = 0.8. The figure shows that the near-miss clone pairs detected by XIAO are a significant portion of all the clone pairs, ranging from 63% to 93% for the commercial codebases.

On each of two commercial codebases (out of the seven) at Microsoft, one of its Microsoft engineers (i.e., those who developed the codebase and are familiar with the codebase) helped evaluate some clone-analysis results generated by XIAO on the codebase. We named these two engineers as Engineers I and II.

[2] An immune clone is one of no particular interest to engineers.

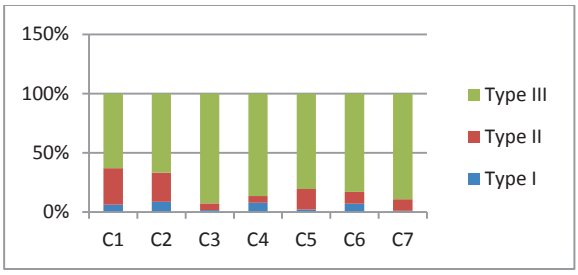


Figure 6. Distribution of clone types of seven commercial codebases (all in C/C++ except C1 in C#) detected by XIAO

Engineer I reviewed 69 clone groups (each of which includes a set of similar clone pairs) with 184 functions in total. All reviewed

```
    // 14 identical statements omitted here
    ::SendMessage(hwndCombo, CB_LIMITTEXT, GetMaxCharacters(),
0);
    int iFlags = 0;
    if (!GetIsIMEAvailable())
    iFlags |= SES_NOIME;
    if (iFlags)
    ::SendMessage(hwndCombo, EM_SETEDITSTYLE, iFlags, iFlags);
    // 5 identical statements omitted here
    ::SendMessage(hwndCombo, EM_SETCOMBOBOXSTYLE,
    SCB_NOAUTOCOMPLETEONSIZE, SCB_NOAUTOCOMPLETEONSIZE);
    // 2 identical statements omitted here
```

Figure 7. A confirmed bug: extra statements for bug fixing were added (with the gray background) to one function but not to its cloned one.

functions are of non-zero bug likelihood and refactoring likelihood. Using the tagging functionality of XIAO, Engineer I tagged 7 (10%) clone groups as potential bugs and 16 (23%) clone groups as refactoring targets. All together there were 23 (33%) clone groups that were identified as actionable (i.e., either potential bugs or refactoring targets).

Engineer II evaluated a small set of clones found by XIAO in a system component that consists of high-quality source code. The source code of this component has been stable with few changes for a number of years. We did not expect to find clone-related bugs in this case. Instead, we were interested in looking for refactoring targets in high-quality code. Engineer II reviewed a total of 39 clone groups with 102 functions. The numbers of clones in these clone groups vary from 2 to 7, except one clone group, which contains 20 clones. All these 20 clones deal with Windows Event operations and they have slight differences in code logic. Including this clone group, Engineer II tagged 8 (16.3%) clone groups with 46 functions as refactoring targets.

5.2 Runtime Cost and Scalability

The running time of XIAO against a large codebase (with the default environment) varies depending on the used settings: from 6 minutes (*MinS* = 20, $\alpha = 1$, $\gamma = 1$) to 23 minutes (*MinS* = 10, $\alpha = 0.4$, $\gamma = 0.8$). Basically, increasing γ tends to linearly decrease the spent time; increasing *MinS* decreases the spent time; increasing α does not change the spent time. This behavior can be easily explained: increasing the value of γ leads to a smaller number of clone-candidate functions in the coarse-matching step, thus decreasing the time spent in each of the successive steps; increasing *MinS* leads to a smaller number of snippets to be checked; α is used in the pruning step, which is the last step, and affects only the number of obtained clones but does not affect the spent time.

Instead of using the default environment, we evaluated the scalability of our XIAO system using an HPC cluster with one master node and four computing nodes (a high-performance computing environment that XIAO leverages to deal with a huge number of lines of code). The master node has four AMD Opteron 880 Dual-core 2.4GHz CPUs and 32GB memory. Each of the four computing nodes has two Intel E5335 Dual-core 2.0GHz CPUs and 8GB memory. Both the master and computing nodes are running on Windows Server 2008 HPC Edition.

Online clone search. We indexed a commercial codebase with about 130 million lines of code to evaluate the scalability of XIAO's online clone-search engine. Code snippets with three or more statements are accepted as valid input for clone search. The preprocessing (including source-code parsing, tokenization, and indexing) was conducted on one computing node and it took 3 hours and 42 minutes to finish. Source code is divided into partitions each with 5MB storage size and these partitions are evenly distributed on the four computing nodes. Then 16 instances of the online clone-search engine were running to serve online queries. We randomly selected 1000 code snippets from the codebase as inputs. The size of these 1000 snippets ranges from 3 to 100 and the number of snippets for each size is about the same. The clone-similarity threshold is set to be 0.6. The number of found cloned snippets ranges from 1 to 1000 and the average time of each query is a number of seconds.

Offline clone detection and analysis. We evaluated the performance of XIAO's offline clone detection and analysis on a commercial codebase with 26 million lines of code using the same system setup as that in the online-search environment. Clones of functions with at least 20 lines of statements were found using the similarity threshold of 0.6. Preprocessing was conducted on one computing node. The clone detection and analysis were performed in parallel on the 4 computing nodes. It took 3 hours and 30 minutes to finish the entire process. The time breakdown of each step (in the unit of seconds) is preprocessing (1,014), coarse matching (9,803), fine matching (213), and clone analysis (1,462). The average amount of memory used by each instance of clone detection and analysis is about 120MB.

6. APPLICATION SCENARIOS IN PRACTICE

We have released XIAO inside Microsoft for different development teams to use (with the first version released in April 2009). There were more than 750 downloads of the tool as of the end of year 2010.

Copy-Paste-Bug Detection and Refactoring. An example application scenario of XIAO was already described in Section 1. In this scenario, an engineer at Microsoft reviewed 69 clone groups for a total of 184 snippets taken from the results of code-clone detection for a commercial codebase. All reviewed clones were near-miss code clones. He identified 7 (10%) clone groups as potential bugs and 23 (33%) clone groups as refactoring targets (including the 7 with potential bugs). The motivating example shown in Figure 1 is one of these seven cases. Function A on the left side is from a shared component, and function B on the right side is from an application. As confirmed by the code owner, B was copied from A for quick reuse quite some time ago. However, the engineer of B was not aware of the changes made to A after the copying.

The clone-related bug shown in Figure 7 is another example reported by the same engineer. In this case, the two functions originally had similar functionalities. Later on a number of statements were added to one function (with the gray background in Figure 7) to ensure the synchronization between Windows GDI objects; nevertheless, this bug fix was not applied to the other function.

The two functions shown in Figure 8 have only slight differences. In fact, they are the same except for one similar-logic block (the second statement in the figure) and one different-logic block (the first statement). This case was analyzed by XIAO to have a high rank in both bug likelihood and refactoring likelihood. As confirmed by its engineer, the differences between the two functions are by-design, and the clones are not buggy. In the meantime, this case was confirmed to be refactorable.

Figure 9 shows a clone group that was tagged as "Immune". Although there do exist slight logic differences between the two functions, the differences were confirmed to be intentional. Currently it is difficult for XIAO to handle false-positive cases such as this one in clone analysis.

Based on our observation, an engineer often tries to prioritize his refactoring efforts, i.e., starting from easy-to-refactor clones (which are often those with high similarity). Another factor for tuning parameters is that a higher value of the similarity threshold needs less running time to get clone-detection results. Therefore, the engineer could choose relatively high similarity threshold first (e.g., 100% the same), to get some easy-to-refactor clones within relatively short clone-detection time. If there is a need to aggressively identify more refactoring opportunities, a relatively small value of the similarity threshold could be used. In some situations, a relatively high value of the similarity threshold would be used. For example, we observed that engineers dealing with a codebase with 20+ million LOC would like to identify file-level clones with 99% similarity and set a relatively high value of the similarity threshold to accomplish this goal.

Detection of Duplicated Vulnerable Code. A security-engineering team at Microsoft has been using XIAO's online clone-search service since May 2009 to help with their investigation on security bugs. There were more than 590 million lines of code being indexed. During the second half of year 2010, there were a number of vulnerable code snippets searched against the XIAO service. Among these searching cases, there were 18.3% cases with good hits, i.e., for these cases, the security-engineering team needs to do further investigation to confirm whether there are duplicated vulnerabilities. Given high severity of security bugs, 18.3% good-hit cases are very good results.

In an example real case, a reported security vulnerability could cause potential heap corruption and lead to remote code execution. After investigation, the vulnerable code snippet was found in codebase A: a buffer-overflow check was missing there.

Using XIAO's clone-search service, one security engineer on the security engineering team found three clones of the vulnerable code snippet – one is also in codebase A and the other two belong to codebase B. This security engineer contacted the code owners of these three cloned snippets and confirmed that one snippet in codebase B was vulnerable. After the contact, the development team owning the vulnerable cloned snippet in B had confirmed to fix this security bug while the security bug in codebase A was fixed.

XIAO's clone-search service has greatly improved the productivity of the security engineers and it enhanced the reliability of the bug-investigation process as well. Based on the clone-search results, security engineers are able to obtain a better understanding of the potential impact of security vulnerabilities and communicate more effectively with development teams on vulnerability investigation and fixing.

In this application scenario of XIAO, security engineers would like to have high recall of clone detection (i.e., little chance of missing clones). Therefore, for this application scenario, XIAO has the default value of 0.6, a relatively small value for the similarity threshold. The value is tunable by security engineers to achieve even higher recall.

Discussion. For the two types of application scenarios, we observed that the second scenario on detecting duplicated vulnerable code (with the target users as security engineers) has occurred much more often than the first scenario, especially on refactoring (with the target users as software engineers). Such observation could be explained with two factors. First, refactoring conducted by software engineers occurs much less frequently than investigation of security bugs, which are the routine work of security engineers. Second, the severity of consequence on missing a refactoring opportunity is much less than the one on missing a security bug.

```
    // 6 identical statements omitted here

    RectF rectImage(0.0f, 0.0f, (float)m_piISGU-
>GetItemWidthPx() - 1.0f, (float)m_piISGU->GetItemHeightPx() -
1.0f);

    // 61 identical statements omitted here
    colorBorder.SetFromCOLORREF(GetBorderColor());

    // 2 identical statements omitted here
```

```
    // 6 identical statements omitted here

    RectF rectImage(0.0f, 0.0f, (float)s_cxInkItem - 1.0f,
(float)s_cyInkItem - 1.0f);

    // 61 identical statements omitted here
    colorBorder.SetFromCOLORREF(GetFrameColor());

    // 2 statements identical omitted here
```

Figure 8. A confirmed example of code refactoring

```
if (!pxdsi || !pxdsl)

// 13 identical statements omitted here
if (FAILED(pxdsi->HrDeleteNode(ppxslChildren[l])))

// 10 statements identical omitted here
```

```
if (!m_spxdsi || !m_spxdsl || !m_pDesc)

// 13 identical statements omitted here
if (!ParseProperty(ppxslChildren[l]))

// 10 identical statements omitted here
```

Figure 9. A clone group tagged as "Immune"

7. RELATED WORK

Research on code-clone detection has been an active research topic in recent years [3][10][17][24][27]. Roy et al. [27] conducted an extensive survey on this research topic.

In contrast to other previous approaches on code-clone detection that conduct aggressive code parameterization without imposing any constraint on characteristics of statements (e.g., CCFinder [18], CP-Miner [21], and Deckard [14]), our code-similarity metric enables users to control the degree of tolerating statement variations by parameter α, allowing XIAO to filter out many false-positive clones that other approaches would report. Our code-similarity metric also enables users to control the percentage of inserted/deleted/modified statements, allowing XIAO to detect near-miss code clones with any number of statement gaps. At the same time, the algorithm efficiency is still achieved since XIAO uses a coarse-to-fine mechanism. Token-based approaches either cannot effectively detect near-miss clones (e.g., CCFinder) or cannot efficiently detect clones with over three gaps (e.g., CP-Miner).

Clone-detection approaches based on parse tree (e.g., CloneDR [5][6] and Deckard) can detect near-miss clones with over three-statement gaps. However, in their approaches, either the percentage of shared tokens [5][6] or the feature-vector distance [14] is used to approximate the tree-edit distance. Although such approximation enables efficient detection algorithms, it leads to false positives, due to the loss of structural similarity caused by the approximation.

Our code-similarity metric also takes into account disordered statements, allowing XIAO to detect near-miss clones with disordered statements. Many other token-based detection approaches such as CCFinder or CP-Miner do not detect clones with disordered statements; parse-tree-based approaches can detect clones with disordered statements; however, they suffer from false positives.

Recently, Gabel et al. [11] proposed a scalable algorithm for detecting semantic code clones based on dependency graphs. They defined semantic code clones as isomorphic sub-graphs of the code's dependency graph. Kim et al. [19] also proposed a memory-comparison-based algorithm for code-clone detection, called MeCC. Their approach can detect near-miss code clones, including clones with disordered statements. Their focus is on detecting semantic code clones, and it is unclear how their detected code clones overlap with near-miss code clones (the focus of XIAO). Such investigation is left for future work.

Besides advances in clone detection, recent research has also made progress on applying clone detection in various software-engineering tasks such as bug detection and refactoring. Near-miss code-clone detection has been used to help identify code-refactoring opportunities [12][31] or find plagiarisms [25][26]. To search whether there are cloned copies of a piece of buggy code, Li and Ernst proposed CBCD [23], a scalable clone-search algorithm that compares graph isomorphism over program dependency graphs. At Microsoft, XIAO has also been used for searching cloned code (e.g., detection of duplicated vulnerable code) and finding refactoring opportunities; comparing to these previous approaches, XIAO is more general and can be used in broader scenarios with high tunability, scalability, compatibility, and explorability.

One important application of detecting near-miss code clones is helping engineers to identify potential bugs caused by inconsistent code changes. CP-Miner [21] detects bugs caused by inconsistently renamed identifiers. The approach by Jiang et al. [15] detects inconsistent contexts of detected code clones. Since XIAO is able to detect near-miss code clones with arbitrary gaps, XIAO has the capability of detecting more types of bugs caused by inconsistent code changes.

There are various tools for code-clone detection available as either open-source tools or commercial tools. Each one performs well in only some aspects. Most of them can detect type-I/II clones well, but have limited capability on detecting type-III clones. Few of them can detect code clones with disordered statements. Few of them provide good tunability. In contrast, XIAO can detect type-III code clones with or without disordered statements, and has high tunability on the tolerance of inserted/deleted statements.

Some of existing tools provide Graphical User Interfaces (GUI) available for exploring code clones. There exists a GUI front-end called GemX for CCFinder [18] to allow users to interactively explore clones with different metrics, such as LOC and distance of folder locations. CP-Miner provides visualization for highlighting clone differences without the concept of blocks. Simian[3] is a Similarity Analyzer for identifying duplication in code written in various languages. It provides limited explorability, displaying only one snippet from each clone group (assuming all copies from a clone group are exactly the same). CloneDR [5][6] provides a summary report and individual clone-set reports, but provides no visualization of clone differences. The uniqueness of XIAO in terms of explorability lies in supporting rich interaction and visualization: intuitive visualization of differences between cloned snippets besides allowing users to tag code clones.

There are some available tools with features of code-clone management, such as CloneTracker [9] and SimScan [4]. CloneTracker is useful for engineers to track code clones. SimScan also provides GUI for clone management and tracking, supporting simultaneous editing. XIAO's tagging mechanism can serve for similar purposes but XIAO provides both clone detection and management with high tunability, scalability, compatibility, and explorability.

The most recent related work is the work done by Jang et al. [13]. They developed a scalable approach for detecting unpatched code clones. Their approach is language agnostic and produces relatively low false-detection rate. They applied their approach on entire OS distributions. While sharing the features of high scalability and compatibility as their approach, our approach is applied on commercial codebases, and is designed to be continuously used by engineers in their daily practices. Therefore, our approach has unique features such as high tunability and explorability.

8. CONCLUSION

In this paper, we report our experiences of carrying out successful technology transfer of our new approach of code-clone detection, called XIAO. XIAO has been integrated into Microsoft Visual Studio 2012, to be benefiting a huge number of engineers in industry. We have discussed main success factors of XIAO: its high tunability, scalability, compatibility, and explorability. We have also presented empirical results on in-practice applying XIAO on real scenarios within Microsoft for the tasks of security-bug detection and refactoring. The results demonstrate the

[3] http://www.harukizaemon.com/simian/index.html

[4] http://blue-edge.bg/simscan/

benefits of XIAO in these tasks. In addition, it was observed that applying XIAO on detecting duplicated vulnerable code (with the target users as security engineers) has occurred much more often than the applying XIAO on refactoring (with the target users as software engineers).

9. ACKNOWLEDGMENTS

We thank our (former) colleagues and interns for their contribution on the implementation of XIAO: Sanhong Chen, Yan Duan, Tiantian Guo, Shi Han, Ray Huang, Qi Jiang, Feng Li, Xiujun Li, Jianli Lin, Huiye Sun, Jinbiao Xu, Jiacheng Yao, and Chiqing Zhang. We thank our colleagues for their help and joint efforts on the successful tech transfer of XIAO, especially Ian Bavey, Gong Cheng, Sadi Khan, Weipeng Liu, and Peter Provost. We thank our colleagues at Microsoft for their feedback and discussion, especially Jonus Blunck, Andrew Fomichev, Shi Han, Xiaohui Hou, Peter Nobel, Landy Wang, Jinsong Yu, and Qi Zhang. We also thank Simone Livieri for his help on evaluations of XIAO.

10. REFERENCES

[1] http://en.wiktionary.org/wiki/inversion_pair, as of Feb. 26, 2011.

[2] http://www.slideshare.net/icsm2011/lionel-briand-icsm-2011-keynote

[3] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proc. WCRE*, pages 86–95, 1995.

[4] I. D. Baxter, C. Pidgeon, and M. Mehlich. DMSR: Program transformations for practical scalable software evolution. In *Proc. ICSE*, pages 625–634, 2004.

[5] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proc. ICSM*, pages 368-377, 1998.

[6] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, E. Merlo. Comparison and evaluation of clone detection tools, TSE, 33(9):577-591, 2007.

[7] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. R. Engler: A few billion lines of code later: using static analysis to find bugs in the real world. Commun. ACM 53(2):66-75, 2010.

[8] Y. Dang, S. Ge, R. Huang, and D. Zhang. Code clone detection experience at Microsoft. In *Proc. IWSC*, pages 63-64, 2011.

[9] E. Duala-Ekoko and M. P. Robillard. Tracking code clones in evolving software. In *Proc. ICSE*, pages 158-167, 2007.

[10] M. Gabel, J. Yang, Y. Yu, M. Goldszmidt, and Z. Su. Scalable and systematic detection of buggy inconsistencies in source code. In *Proc. OOPSLA*, pages 175-190, 2010.

[11] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *Proc. ICSE*, pages 321-330, 2008.

[12] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. ARIES: Refactoring support tool for code clone. In *Proc. WoSQ*, pages 1-4, 2005.

[13] J. Jang, A. Agrawal, and D. Brumley. ReDeBug: Finding unpatched code clones in entire OS distributions. In *Proc. S&P*, pages 48-62, 2012.

[14] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. *Proc. ICSE*, pages 96-105, 2007.

[15] L. Jiang, Z. Su, and E. Chiu. Context-based detection of clone-related bugs. In *Proc. ESEC/FSE*, pages 55-64, 2007.

[16] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, Do code clones matter? In *Proc. ICSE*, pages 485-495, 2009.

[17] E. Juergens and N. Göde. Achieving accurate clone detection results. In *Proc. IWSC*, pages 1-8, 2010.

[18] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. TSE, 28(7):654–670, 2002.

[19] H. Kim, Y. Jung, S. Kim, and K. Yi. MeCC: Memory comparison-based clone detector. In *Proc. ICSE*, pages 301-310, 2011.

[20] R. Koschke. Survey of research on software clones. In *Proc. Duplication, Redundancy, and Similarity in Software*, 2006.

[21] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. *Proc. OSDI*, pages 289–302, 2004.

[22] M. Li, J. Roh, S. Hwang, and S. Kim. Instant code clone search, In *Proc. ESEC/FSE*, pages 167-176, 2010.

[23] J. Li and M. D. Ernst. CBCD: Cloned buggy code detector. In *Proc. ICSE*, pages 310-320, 2012.

[24] S. Livieri, Y. Higo, M. Matushita, and K. Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed CCFinder: D-CCFinder. In *Proc. ICSE*, pages 106-115, 2007.

[25] L. Prechelt, G. Malpohl, and M. Philippsen. JPlag: Finding plagiarisms among a set of programs. Technical report, University of Karlsruhe, Department of Informatics, 2000.

[26] R. Robbes, R. Brixtel, M. Fontaine, B. Lesner, and C. Bazin. Language-independent clone detection applied to plagiarism detection. In *Proc. SCAM*, pages 77-86, 2010.

[27] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. In Science of Computer Programming, 74(7):470-495, 2009.

[28] R. Tiarks, R. Koschke, and R. Falke. An assessment of type-3 clones as detected by state-of-the-art tools. In *Proc. SCAM*, pages 67-76, 2009.

[29] Y. Ueda, T. Kamiya, S. Kusumoto and K. Inoue. Gemini: Maintenance support environment based on code clone analysis. In *Proc. IEEE METRICS*, pages 67-76, 2002.

[30] M. J. Wise. String similarity via greedy string tiling and running Karp-Rabin matching. Department of Computer Science, University of Sydney, ftp://ftp.cs.su.oz.au/michaelw/doc/RKR GST.ps, December 1993.

[31] L. Yu and S. Ramaswamy. Improving modularity by refactoring code clones: A feasibility study on Linux. In SIGSOFT Notes, 33(2), 2008.

Self-Healing Multitier Architectures using Cascading Rescue Points

Angeliki Zavou, Georgios Portokalidis, Angelos D. Keromytis
Department of Computer Science
Columbia University, New York, NY, USA
{azavou, porto, angelos}@cs.columbia.edu

ABSTRACT

Software bugs and vulnerabilities cause serious problems to both home users and the Internet infrastructure, limiting the availability of Internet services, causing loss of data, and reducing system integrity. Software self-healing using rescue points (RPs) is a known mechanism for recovering from unforeseen errors. However, applying it on multitier architectures can be problematic because certain actions, like transmitting data over the network, cannot be undone. We propose *cascading rescue points* (CRPs) to address the state inconsistency issues that can arise when using traditional RPs to recover from errors in interconnected applications. With CRPs, when an application executing within a RP transmits data, the remote peer is notified to also perform a checkpoint, so the communicating entities checkpoint in a coordinated, but loosely coupled way. Notifications are also sent when RPs successfully complete execution, and when recovery is initiated, so that the appropriate action is performed by remote parties. We developed a tool that implements CRPs by dynamically instrumenting binaries and transparently injecting notifications in the already established TCP channels between applications. We tested our tool with various applications, including the MySQL and Apache servers, and show that it allows them to successfully recover from errors, while incurring moderate overhead between 4.54% and 71.56%.

Categories and Subject Descriptors

D.4.5 [**Software**]: Operating Systems—*Reliability*

General Terms

Reliability, Security

Keywords

Software self-healing, error recovery, reliable software, multitier applications

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

1. INTRODUCTION

Software bugs and vulnerabilities cause serious problems to both home users and the Internet infrastructure. Such problems include broad outages [23], integrity violations [26], and data loss [14]. Despite the great combined efforts of both industry [12] and researchers [4, 8] the continuously increasing size and complexity of software makes it extremely difficult to produce error-free software. To mitigate the effects of bugs that can reduce the integrity of systems, a plethora of runtime protection mechanisms have been devised, like stack smashing protection [10], write integrity testing [2], address space layout and code randomization [22, 21, 27]. Nevertheless, while protection mechanisms render certain types of vulnerabilities infeasible or impractical, they do not also offer high availability and reliability, as they frequently resort to terminating applications that behave abnormally to prevent attackers from performing any useful action.

To increase software availability, many mechanisms that aim to recover execution when unhandled errors occur have been proposed [16]. One of these mechanisms is software self-healing based on rescue points [30]. It operates based on the observation that applications already contain code for handling anticipated errors and proposes reusing this code to also handle unexpected errors. Rescue points (RPs) are essentially functions that contain error handling code, which can be exploited to recover from errors occurring within the RP, including the RP routine itself and all called routines. A checkpoint is taken upon entering a RP, and execution is rolled back to that checkpoint when an unhandled error occurs, while concurrently a valid error code is returned by the RP to the application (*i.e.,* through the routine's return value), so that it can gracefully handle the failure.

Applying RP-based self-healing on self-contained functions is straightforward, however there are many functions that have side effects, such as transmitting data to other entities on the network. Applications that are part of multitier architectures, like client-server or three-tier architectures (comprised by presentation, logic, and data tiers), contain many such functions. Introducing RPs in such architectures can be problematic because it can result in inconsistent states between the tiers when a roll back occurs. For example, consider the following. The first tier communicates certain information to the second tier, which then communicates with the third tier, and so on. If an error occurs in the first tier, triggering a rescue point, the application will think that an error, like a communication failure has occurred, while in fact the effects of the transmission have already propagated to other tiers.

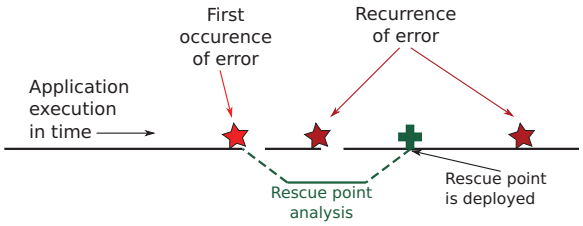


Figure 1: Software self-healing transforms unanticipated errors from fail-stop to fail-once. After an unexpected error first occurs, causing the application to terminate, the data produced during the fault (*e.g.,* a core dump) are used to analyze the fault and produce a remedy in the form of a rescue point. While the application is still "vulnerable" until the (either automatic or manual) offline analysis is completed, after the rescue point is deployed, a recurrence of the fault will be gracefully handled.

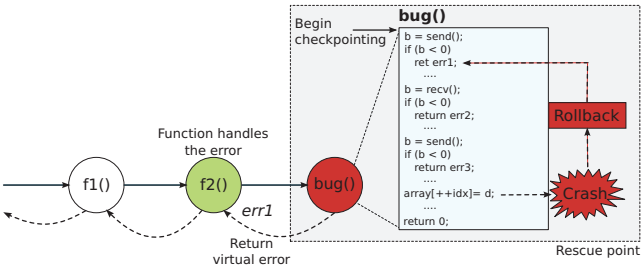


Figure 2: Software self-healing using rescue points. Function *bug()* contains an error which can cause an application crash. If it, or a caller function (*e.g., f1()* and *f2()*), contains error handling code for expected faults, it can be used to handle unexpected errors, *i.e.,* it is a possible *rescue point.* A checkpoint is made upon entering the rescue point, and execution is rolled back when an error occurs. We return a valid error code to allow the application to continue executing (dashed arrows).

We propose *cascading rescue points* (CRPs) for self-healing applications in multitier architectures to address the inconsistency issues introduced by traditional RPs. In our approach, when an application executing within a RP communicates with an application on the next tier, we notify the remote peer to also perform a checkpoint, cascading, in this way, the checkpoint and RP to the lower tiers of the architecture. If a RP successfully completes execution or if it triggers a roll back due to an error occurring, a notification is also send to all the peers that were instructed to checkpoint, so that they also perform the appropriate action.

We have implemented CRPs using the Pin [18] dynamic binary instrumentation framework for x86 Linux, extending our previous work [28] on deploying traditional RPs using Pin. We improve the checkpointing mechanism used by utilizing the *fork()* system call to quickly create copy-on-write copies of an application's image and use filters to mark the individual bytes modified by threads for efficient thread-wide checkpointing. We also intercept system calls to restore the contents of overwritten memory and to transparently inject information in the communication channels between applications of different tiers that run on top of our tool. We use the injected data to implement a protocol for conveying notifications between the various parties. Additionally, we utilize TCP out-of-band data to asynchronously notify remote peers of a successful exit from a RP.

In practice, we envision RPs being employed as a temporary solution for running critical software until a concrete solution, in the form of a dynamic patch or update, is available. Using a dynamic framework like Pin enables us to attach and detach our tool on already running applications without interrupting its operation, applying RPs only for as long as they are required. Combined with a dynamic patching mechanism [7, 11, 19], applications can be run and eventually patched without any interruption.

Distributed checkpointing and recovery has been a popular subject of research [5, 32]. However our work is driven by other goals and differs from previous work in the following ways:

- Our approach is *transparent* and *self-contained*. It does not require that applications are designed with self-healing in mind, nor does it require support from the operating system, and it is applicable on binary-only software
- We do not checkpoint at arbitrary points of execution, but instead checkpointing is driven by rescue points
- We can dynamically engage/disengage software self-healing to apply it only when needed
- Our tool piggybacks the checkpointing protocol on existing communication channels

We evaluate our approach using popular servers applications, like Apache and MySQL, that suffer from well known vulnerabilities and show that our CRP protocol does not introduce prohibitive overheads. The performance overhead imposed by our approach varies between 4.54% and 71.96% depending on the application. Note that our approach can be ported with moderate effort to operate on other platforms supported by Pin, including Windows and BSD operating systems, and the x86-64 architecture.

This paper is organized as follows: Section 2 contains some background information on the tool we use for developing CRPs, and discusses the limitations of traditional RPs. An overview of cascading rescue points is given in Sec. 3. We describe the implementation of a prototype in Sec. 4, and evaluate its effectiveness and performance in Sec. 5. Related work is discussed in Sec. 6. We conclude in Sec. 7.

2. BACKGROUND

2.1 Software Self-healing Using Rescue Points

The goal of software self-healing is to allow applications to operate normally by healing themselves when unanticipated errors occur. ASSURE [30] was one of the first works to present a practical and automatic approach to software self-healing. Fig. 1 depicts a high level overview of the concept. When an error first occurs, it is analyzed offline to determine its location and the appropriate remedy to be applied that will allow the application to self-heal when it reoccurs.

One of the key ideas of software self-healing is rescue points (RPs). RPs are essentially routines that contain

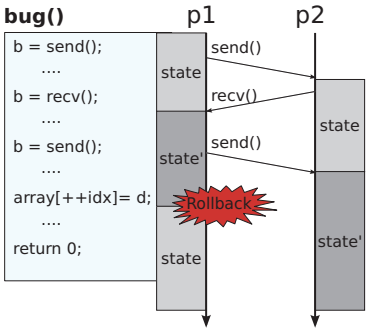


Figure 3: A rescue point deployed on function *bug()* of process $p1$ needs to both send and receive data to and from $p2$. When an error triggers a roll back, $p1$ can end up in an inconsistent state with $p2$. Deploying rescue points in routines that communicate with other parties over the network can be problematic because their effects cannot be reversed.

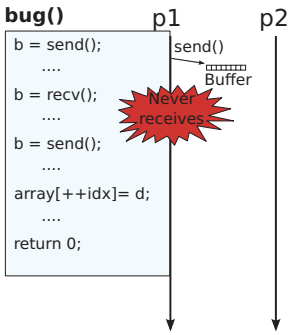


Figure 4: Adopting a naive approach to address the issue in Fig. 3 will not work. For example, buffering the data being send from a rescue point, and only transmitting them after determining that an error did not occur, can break applications. In this case, $p2$ never receives the data that will cause it to respond to $p1$.

error handling code written by the programmer to handle expected error conditions, and directly or indirectly (*e.g.,* through a function call) engulf code containing an unexpected fault. ASSURE proposed the use of existing error handling code to gracefully handle unanticipated faults, virtualizing in this manner error handling, by mapping the larger set of unknown errors that can occur during execution (*e.g.,* invalid memory accesses and attacks) to the smaller set of handled errors (*e.g.,* a system call failing).

2.1.1 Discovering Rescue Points

ASSURE proposed a mechanism for automatically discovering possible RPs and selecting the one that is more likely to patch an observed error. The goal was to identify program functions and returned error codes through fault injection. Consider the function *bug()* in Fig. 2; *bug()* may return *err1* or *err2*, if *send()* or *recv()* fail respectively. This designates the function as a potential RP for errors occurring within the function, or a function that it calls, because it can return a valid error code to *f2()*, allowing it to handle an unanticipated error, such as an out of bounds access of *array* that could cause the application to crash.

The simplest way to detect unknown errors and initiate the rescue point analysis is to intercept the signals (or exceptions in Windows OSs) that are raised when a serious error such as an invalid memory reference occurs. In Linux, such signals include *SIGSEGV* for memory faults, *SIGFPE* for floating point errors like division by zero, *etc.* Software self-healing can be also employed in conjunction with protection mechanisms already incorporated in the application [10, 2, 22], or retrofitted on the binary after it was deployed [27, 15]. For example, ProPolice [10] uses the *abort()* system call, which raises signal *SIGABRT*, when a stack smashing attack is detected.

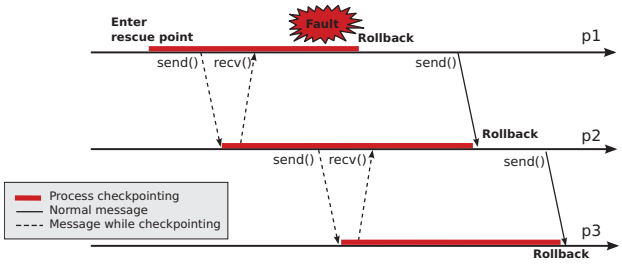
The primary goal of ASSURE was to automate the process of discovering, selecting and deploying an RP, however RPs can be also discovered manually. For example, the operating system can be configured to produce a dump of the memory image of processes crashing due to a memory violation error. This core dump can be manually analyzed by a developer or administrator to determine the location of the error [1] and look for an appropriate RP. While this process is time consuming and requires user intervention, producing and distributing an actual software patch that corrects the error at its source, frequently requires even more time and resources. Security related patches can take as much as two weeks from the date they have been disclosed [3], while less critical faults that only affect the availability of software may take even longer [37].

2.1.2 Rescue Point Deployment

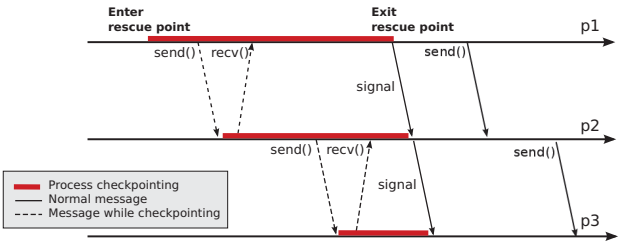
ASSURE relies on process-wide checkpoint/rollback based on Zap [20] to create checkpoints, as well as to rollback to a checkpoint when an error occurs. Because of Zap, the overhead is little, but it is not extremely practical as it requires modifications to the Linux kernel, and cannot be dynamically installed and removed. In previous work [28], we designed and implemented REASSURE a tool that simplified the deployment of RPs. We built upon Intel's Pin dynamic instrumentation framework [18], to create a self-contained mechanism that can dynamically deploy RPs on binary-only software, for as as long as it is required. The use of Pin enables us to attach to an already running application to deploy a RP on demand, while we can also detach from the application to apply a patch at runtime [19]. In this paper, we build on our previous work to enable the application of rescue points on multiparty software systems.

2.2 The Problem: Irreversible Side-effects within Rescue Points

Previous software self-healing approaches cannot apply rescue points on functions that have side effects, such as transmitting data to other entities on the network. Doing so can result in inconsistent states between the communicating parties, as shown in Fig. 3, because the effects of process $p1$ sending a message to $p2$ cannot be undone. The problem with this scenario is that the client's state has been rolled back and the client believes that an error, such as being unable to communicate with the server, occurred. However, data has been exchanged with the server, which is oblivious of the error that occurred in the client. Depending on the nature of the communicating applications this can lead



(a) Fault occurs. The next transmission from *p1* to *p2* will notify the latter to also roll back. *p2* will eventually notify *p3*.



(b) No fault. When *p1* exits the rescue point, it immediately notifies *p2*, which also exits checkpointing and notifies *p3*, and so forth.

Figure 5: Cascading rescue points overview. When process *p2* receives a message from process *p1*, which executes within a rescue point, it also begins checkpointing. Other processes, like *p3*, that receive messages from a checkpointing process also begin checkpointing. This way the original rescue point *cascades* to the communicating processes.

to various problems and can require additional mechanisms, like transactions employed by database (DB) servers, for restoring them to a consistent state. By consistent state, we refer to every party having a correct view of what is the state of their peer. For example, if $p1$ tries to issue a command to $p2$ to switch it to state $state'$ and it fails, $p1$ can still think that $p2$ is in state $state$.

Previous designs simply ignore data exchanges and rely on the protocols implemented by applications to discover and correct such inconsistencies (*e.g.,* they use transactions). Moreover, the problem cannot be trivially addressed by simply delaying the transmission of messages. Fig. 4 depicts an example where the application expects to receive a response to a message send from within a RP. Buffering the transmitted message would break function *bug()*, causing it to fail or wait forever because no data is sent as a response from $p2$.

3. CASCADING RESCUE POINTS

3.1 Overview

Self-healing using cascading rescue points aims to enable applications participating in multitier architectures to self-heal without facing the problems presented in Sec. 2.2. To achieve this, we introduce a protocol, which is transparently implemented over the application's TCP connections. The protocol encapsulates application data, and serves the sole purpose of allowing us to convey signals between applications of the architecture.

Consider process *p1* shown in Fig. 5. All of its communications with other processes in the architecture are modified to implement our CRP protocol. When *p1* executes within a RP, it is essentially checkpointing, indicated by the highlighted areas in Fig. 5. This means that a fault will cause all the changes performed within the RP to be undone, and we will simulate the return of an error code from the RP routine. When *p1* transmits data to another process (while in a RP), we use our protocol to instruct the remote peer to also begin checkpointing. Later on, if an errors occurs in *p1*, the RP will recover the process. Since we piggyback our protocol on existing communications, *p1* does not immediately notify *p2* that it discarded the state generated in the RP, and *p2* will continue checkpointing until the next message is received by *p1*. Figure 5(a) depicts this process, which is propagating in time to the other processes. If *p2* sends any data to another process (*e.g., p3*), that process also begins checkpointing, and so forth (see Sec 3.2 for limitations).

If no fault occurs, the process is almost entirely the same, and it is shown in Fig. 5(b). Like before, the RP of *p1* causes the checkpointing to cascade to *p2* and *p3*. However, in this case no error occurs and *p1* successfully exits the RP. When this happens, we immediately notify the other processes by utilizing TCP's out-of-band (OOB) data [33]. OOB data are not part of the regular data stream, so we can signal *p2* and *p3* without corrupting the application data stream and without requiring data to be read by the processes. Instead, we can rely on the OS to notify the process when such a packet is received (*e.g.,* by raising a signal or exception). TCP does not support multiple OOB signals on a particular stream (*i.e.,* a second OOB would overwrite the first and would be the only one to raise a signal on the receiver). For this reason, we can only use it to signal successful exits from RPs. Our approach is an optimistic one, assuming that errors will be rare.

We are planning to explore scenarios with more frequent errors, *e.g.,* when the application is under attack. One approach we are looking into is to be able to allow our protocol's notification system to adapt depending on the conditions of the involved applications. For example, for processes where errors are too frequent, the CRP protocol could switch the notification methods used for notifying the communicating processes for the events of successful or not exit from an RP. The OOB signal used in the current CRP protocol as to notify the other processes for a successful exit from a RP, could be used for the opposite purpose, *i.e.,* to signal the rest of the processes to rollback to a previous state. Consequently, the next transmission of data would have to piggyback the signal for successful exit from a RP. This new approach requires exploring other important factors first, such as the *necessary* conditions under which the switch of the notification methods would make the CRP protocol more effective.

3.2 Limitations: Overlapping Checkpoints

Our approach enables multiple processes in a multiparty architecture to checkpoint in a loosely coordinated way. However, our goal is *not* to provide another algorithm for coordinated checkpoint/restart for an unstructured distributed or peer-to-peer system. *Our approach is a good fit for archi-*

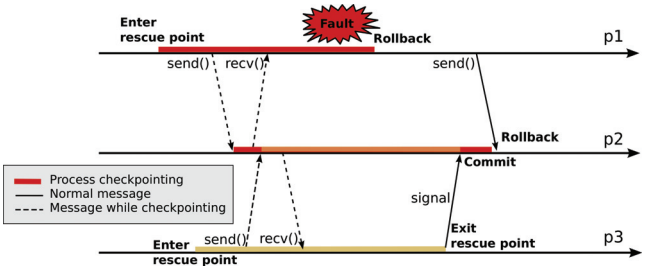


Figure 6: Overlapping checkpoints. Processes *p1* and *p3*, while both in RPs, try to *cascade* their RP to the receiving process *p2*. *p1* requests a rollback, while *p3* commits. The changes made by the latter are lost.

tectures that have some innate hierarchy, like a three-tier or client-server architecture. As it should be already obvious from Fig. 5, receiving a checkpoint request from a process, while already checkpointing due to the request of another process or a RP, has no effect. In this sense, our approach is best-effort and does not offer strong guarantees on establishing a globally consistent state between all processes.

Figure 6 depicts such a case of *overlapping* checkpoints. Two processes *p1* and *p3* enter RPs independently and both transmit data to process *p2*. According to the CRP protocol, *p2* will start the checkpointing, the moment it receives the data from a process in a RP, in this case *p1*. When a process is already checkpointing, it will ignore signals to start checkpointing (*e.g.,* the signal sent from *p3*). We should emphasize that even though we do not handle overlapping checkpoints, their occurrence is not catastrophic to CRPs. However, we do have to rely on application logic to identify and correct errors, which was the case before introducing CRPs.

4. IMPLEMENTATION

4.1 Self-contained Rescue Point Deployment

In previous work [28], we implemented REASSURE a tool for deploying rescue points on binaries in an on-demand fashion and without the need for source code. We built our tool using Pin [18], a framework that enables the development of tools that can at runtime augment or modify a binary's execution at the instruction level through an extensive API. The target binary executes on top of Pin's virtual machine (VM), which essentially consists of a just-in-time (JIT) compiler that combines the binary's original code with the code inserted by the tool and places the produced code blocks into a code cache, where the application executes from. Pin facilitates the instrumentation of binaries by enabling developers to inspect and modify a program's instructions, as well as intercept system calls and signals. It is actively developed and supports multiple hardware architectures and OSs. Pintools can be applied on binaries by either launching them through Pin or by attaching on already running binaries. The latter behavior is highly desirable, as it allows us to deploy RPs without interrupting an already executing application. Our tool currently runs on x86 Linux systems, however there are no significant challenges in porting it to other OSs and architectures supported by Pin.

Table 1: Example of rescue points covering known bugs on popular applications (also see Tab. 2).

Application	Function name/Return value
MySQL v5.0.67	Item_func_set_user_var::update()/1
Apache v1.3.24	ap_bread()/-1

RPs can be installed on any callable application function. Table 1 lists RPs for a set of popular applications, including the error codes that should be returned when an unexpected error occurs. When a RP function is first entered by an application thread, we use Pin's API to insert code that will switch the thread into *checkpointing mode* and save CPU state. Instructions that can be used to exit the RP, such as the function return *RET* instruction on the x86 instruction-set, are also instrumented to return the thread exiting the rescue point to *normal mode* and to discard the previously saved state.

When executing in checkpointing mode, our tool instruments all memory write instructions and logs the overwritten memory contents into a dynamically expanding array, the *write log*. Pin provides us with facilities so that only the instructions being reached from within a RP are actually instrumented in this fashion. This way individual threads of an application can enter RPs and checkpoint individually (assuming that no shared state is updated).

To identify errors our tool relies on signals. In particular, we intercept the following signals on Linux: *SIGSEGV*, *SIGILL*, *SIGABRT*, *SIGFPE*, *SIGPIPE*.[1] When a thread executing within a RP receives one of these signals, we initiate the recovery process. The recovery process restores memory contents and execution returns to the callee, also returning the error code specified by the RP. In x86 architectures, function return values are usually returned in the *EAX* register.

Concurrency.

Checkpointing is *thread-specific*, that is multiple threads can enter a RP at the same time and each thread can roll back independently. However, if a RP is processing data shared by multiple threads, or if the error that causes the application crash corrupts data used by other threads, since they are all executing in the same address space, this type of checkpointing can leave the process in an inconsistent state after a roll back. We address such issues by introducing *blocking* RPs that block other threads for their duration. We can block threads by conditionally instrumenting every block of instructions, so that when a certain flag is asserted execution is blocked. This *always-on blocking* approach is appropriate for applications that expect a very high rate of faults. Alternatively, we utilize signals to asynchronously interrupt the remaining threads of a process and block them until execution has left the RP. This *on-demand blocking* mode generally incurs less overhead, since no additional instrumentation needs to be added for non-RP code.

[1]Note that other OSs have similar mechanisms to synchronously notify applications of such errors. For example, Windows OSs use exceptions.

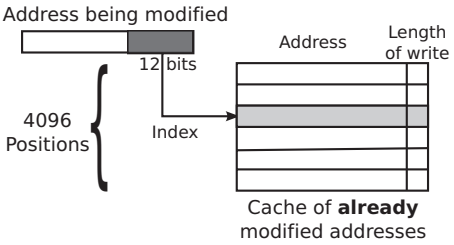


Figure 7: A small associative cache is used to quickly check if a memory location has been already modified. The cache is indexed using the lower 12-bits of an address. Each slot stores the address and number of bytes already modified.The original and cached addresses, as well as the length of the write, are compared to determine a cache hit. A cache miss updates the appropriate slot.

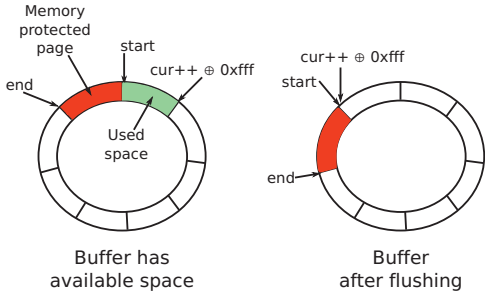


Figure 8: A circular buffer can be used to store the memory locations modified by a thread during checkpointing. When full, a page protection fault is generated. We capture the fault to flush the buffer contents to disk, and setup a new protected page. The failed insertion can then simply resume to be completed.

4.2 Efficient Thread Checkpointing

Checkpointing threads individually is beneficial to software self-healing. If an error occurs while a thread is executing within a RP, we only need to roll back the state of the thread that suffered the fault, while the remaining threads can execute unhindered. However, using a *writes log* to store the overwritten memory values does not scale for certain applications and can lead to excessive memory overheads. *We address these issues by extending our tool in three ways*:

1. We introduce a small associative cache (shown in Fig. 7) to quickly check if a memory location has been already modified

2. We use the *fork()* system call to create a copy-on-write image of the process's address space and employ a *filter* to mark the memory locations updated by the checkpointing thread. Two types filters are currently supported: a statically allocated *bitmap* where each bit corresponds to a four-byte memory area, and a *circular buffer* that stores the modified addresses of memory

3. For the circular buffer, we utilize page protection and intercept OS page-faults to identify when the buffer is full, and write its contents to disk to avoid excessive memory usage

The cache allows us to minimize the number of updates performed in the writes log and the filter. This has the effect of reducing the amount of memory required for checkpointing, as addresses repeatedly written (*e.g.,* stack variables) are only updated once. We use the lower address bits to index the cache to exploit locality in memory accesses.

Checkpointing using *fork()* is not a novel concept [25]. *fork()* is used to cheaply obtain a copy of the memory contents of the entire process. Memory pages are initially shared between the processes, while the OS creates copies of the pages when they are modified. When a RP is entered, a *checkpointing process* is forked. This newly forked process uses a shared memory segment to communicate with the original one and utilizes a semaphore for proper synchronization. It initially sleeps waiting for events from its parent. If the RP successfully exits (no error occurs), the checkpointing process is signaled to exit. Otherwise, it accesses the filter to obtain the addresses that were modified in the original, and uses a pipe (used in UNIX systems for unidirectional inter-process communication) to return the original memory contents to the main process.

When using a bitmap filter, the overhead is low as the bitmap is allocated once on RP entry and updating it is efficient. Using one bit per four memory bytes means that we could erroneously restore a byte that was not overwritten by the current thread. This could lead in memory corruption, if the particular byte was concurrently updated by another thread. Most compilers tend to use four or even eight byte alignment for variables, so in practice such cases should rarely (if ever) occur. Note that we do not address cases that the application itself erroneously implements synchronization primitives, leading to inconsistent updates of shared state.

Using the circular buffer for storing modified addresses does not suffer from such limitations. Moreover, it uses less memory, making it a good fit for highly parallel processes that may have many active RPs concurrently, and it supports 64-bit systems (64-bit address spaces are too large to be covered by a statically allocated bitmap).

Our circular buffer implementation focuses on very fast updates. This is achieved by first aligning its size to a power of two. This allows us to simply increase the cursor (*i.e.,* the index pointing to the first available slot) after inserting data in the buffer, and use a cheap bit masking operation to down cast it to the size of the buffer. Second, we use page protection to signal buffer fullness instead of checking the number of available bytes on each update. This is accomplished by memory protecting the last page (usually 4KB) of the buffer. When an update spills into the protected region, we flush the buffer to disk, remove the page's protection, update the buffer header, and protect the page that is currently last, as depicted in Fig. 8.

Reverting System Call Effects.

Process memory is not only modified by write instructions, but it can be also written by the kernel during a system call. We extended our tool to intercept system calls and mark the memory locations modified by them in the filter holding the altered memory locations. For this purpose, we define a static array for storing system call related metadata, spec-

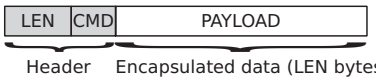


Figure 9: The cascading rescue points protocol encapsulates user data using a small header prepended to every data write made by the user.

ifying the size of their arguments and whether they return data. For example, the *read(int fd, void *buf, size_t count)* system call writes data in the pointer specified by its second argument. The amount of written data depends on its return value. More elaborate calls like *socketcall()* are handled by defining a call back that de-multiplexes the various network calls implemented by it.

4.3 Cascading Rescue Point Protocol

4.3.1 I/O Interception

The cascading rescue point protocol is used to communicate events between peers exchanging data over TCP sockets. The protocol is implemented transparently over the sockets used by the application. This is done by intercepting system calls used with TCP sockets. We can classify these system calls into two groups. The first group consists of calls handling socket creation and termination, and the second group is dealing with data transmission and reception. We intercept system calls *socket()*, *close()*, *shutdown()*, *connect()*, *accept()*, *socketpair()*, and the *dup()* family of calls to track the state of descriptors used by the application (*i.e.,* distinguish TCP socket descriptors from others, like files). For this reason, we maintain a global array to store information on active descriptors, like their type and protocol state data. We also intercept the *read()*, *write()*, *recv()*, and *send()* family of system calls that are used to transmit and receive data from sockets to implement our protocol.

The protocol consists of variable length messages that encapsulate user data as shown in Fig. 9. In particular, we use a small header that comprises of a 4-byte field specifying the length of user data, and a single-byte *CMD* field used to communicate events to remote peers.

The header is inserted into existing TCP streams using Pin to replace system calls used to write data, like *write()* and *send()*, with *writev()* which allows us to transmit data from multiple buffers by performing a single call. This minimizes the number of operations (data copies and system calls) required to transparently inject the header into the stream. If the message cannot be written in its entirety, for instance because non-blocking I/O is performed and the kernel buffers are full, we keep trying until we are successful.

To extract the header from the stream, the reverse procedure is followed. Initially, we replace calls used to receive data with *readv()* to read into multiple buffers. If necessary, we repeat the process until the whole header is received. User data is placed directly in the buffer supplied by the application. However, we can read into the next message, which will be placed into the application's buffer. When this happens, we move the data belonging to subsequent messages into a buffer associated with the socket descriptor. Consequent reads will read data from this buffer instead of performing a system call. Reading one message at a time may be suboptimal performance-wise, but allows us to pair read system calls with particular events received on a socket (*e.g.,* a request to begin checkpointing), which is necessary for rolling back.

Table 2: Applications and benchmarks used for the evaluation of CRP. All of applications contain exploitable bugs as described by their *common vulnerability and exposure* (CVE) id.

Application	Bug type	Benchmark
MySQL v5.0.67	Input validation (CVE-2009-4019)	MySQL's *test-select*
Apache v1.3.24	Memory corruption (CVE-2002-0392)	Apache's *ab* utility

4.3.2 Protocol Commands

The *CMD* field in the protocol is used to inform remote peers of changes in the state of the running thread. For instance, when data are written to a socket while in a rescue point, *CMD* changes to indicate that the destination should also begin checkpointing, the socket is marked as having been signaled to checkpoint, and is placed in a list containing other similar sockets (*fd_checkpointed*). If an error occurs and the thread rolls back, sockets in *fd_checkpointed* are marked accordingly, so that the next write will convey the status change. *If the next write occurs within a RP, the fact is also passed to the remote process, so that it first rolls back memory changes and then enters a new checkpoint.*

On the receiving end, if a thread receives a command to checkpoint, it begins checkpointing similarly to entering a RP. The socket descriptor number where the command was received is saved, so that a consequent request to roll back is only honored, if it was received on the same socket. On rollback, execution resumes right before the system call that caused the thread to checkpoint. Note that receiving requests to begin checkpointing from other sockets, while already checkpointing or executing in a RP are ignored (discussed in Sec. 3.2).

Checkpoint Commits Through Out-of-band Signaling.

To notify remote peers of a successful exit from a RP, we utilize out-of-band (OOB) signaling, as provided by the TCP protocol and the OS. In particular, we make use of TCP's OOB data to notify a remote application that it should also commit changes performed within a checkpoint. We send OOB data by using the *send()* system call and supplying the *MSG_OOB* flag for every descriptor in *fd_checkpointed*.

On the receiver, the reception of an OOB signal by the OS, causes the signal *SIGURG* to be delivered to the thread, which previously took ownership of the socket descriptor that triggered the checkpointing by calling *fcntl()*.[2] The signal is intercepted, and execution is switched from checkpointing to normal execution. If a RP is entered very frequently, multiple OOB signals can be transmitted in succession. On account of TCP's limitations, only a single OOB byte can be pending at any time, so previous OOB signals are essentially overwritten. This does not affect the correct operation of our system, but unfortunately implies that we

[2]In Linux a thread can take ownership of a descriptor, causing the OS to deliver all descriptor related asynchronous events to the specific thread, instead of a randomly selected thread of the process.

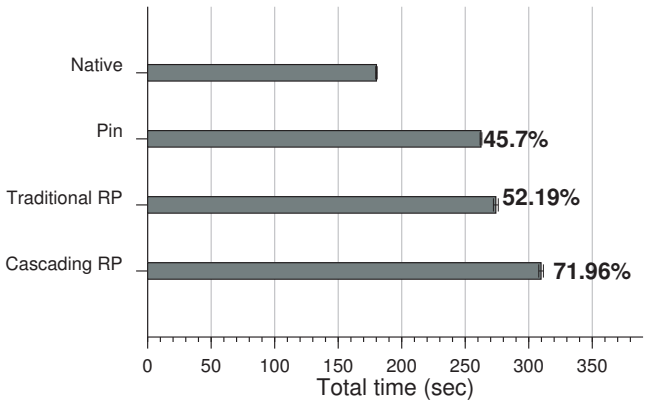


Figure 10: MySQL performance

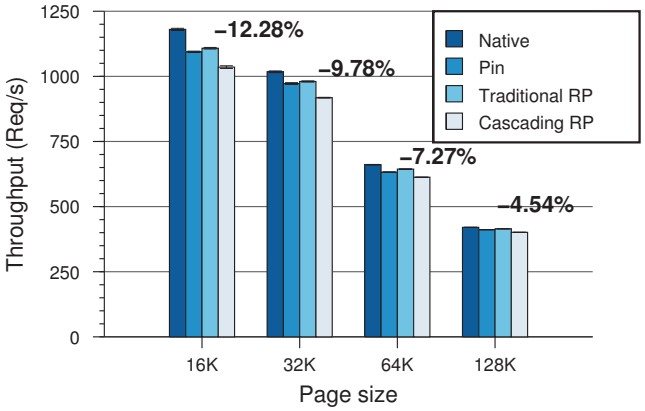


Figure 11: Apache performance

cannot also use OOB signaling to notify remote peers of roll backs.

5. EVALUATION

We evaluated our implementation to establish its effectiveness and performance. First, we validated the effectiveness of CRPs in addressing state inconsistency issues that arise when using RPs to recover from errors in interconnected client-server applications. Second, we evaluated the performance overhead imposed by CRP with real server applications. In both cases, we employed existing benchmarks and tools to generate workloads. Table 2 lists the applications and benchmarks used during the evaluation. We conducted the experiments presented in this section on two DELL Precision T5500 workstations with dual 4-core Xeon CPUs and 24GB of RAM, one running Linux 2.6 and acting as a server and the other one running Linux 3.0 and acting as the client.

5.1 Effectiveness

We used our tool to deploy RPs on known bugs in the applications listed in Table 2, while concurrently running the corresponding benchmarks from the client-side. When RPs are not employed, the applications terminate and the benchmarks are interrupted in all cases. In contrast, when using RPs the applications recover from the error and the benchmarks concluded successfully.

We also used our own artificial client-server applications, that employed our mechanism to *cascade* a RP, which engulfed the exchange of messages between the peers. We manually injected faults in the client and observed that both peers did not crash, but instead they managed to revert to consistent states (*i.e.,* the state they had before entering the RP).

5.2 Performance

For each application in Table 2, we performed the corresponding benchmark, first with the application executing natively, second running under the Pin DBI framework, then employing traditional RPs, and finally with CRPs. This allows us to quantify the overhead imposed by CRP compared with native execution and execution under Pin, as well as the relative overhead compared with the baseline of a self-healing tool with traditional RPs. In the tests described in this section, we did not inject any requests that would trigger the bugs each application suffers from, nevertheless the relevant RPs to the known bugs were installed and in the last case the CRP mechanism was employed.

Figure 10 shows the results obtained after running 10 iterations of MySQL's *test-select* benchmark test over an 1Gb/s network link. The y-axis lists the four different server configurations tested, which from top to bottom are: native execution, execution over Pin, execution with our self-healing tool and traditional RPs, and finally execution with CRPs. The x-axis shows the average time (in seconds) needed to complete the benchmark, while the errors bars represent standard deviation. Note that the standard deviation for is insignificant and thus not visible. We observe that the benchmark takes on average 71.96% more time to complete when running the server with a CRP, while a significant part of the overhead is because of Pin (under Pin the test takes 45.7% more time). In particular, the high overhead is attributed to the significantly larger size of MySQL's code consisting of many indirect control transfers (*e.g.,* callback functions and frequent function return), which exerts pressure on Pin's JIT compiler and code cache.

Figure 11 depicts the results obtained after performing 10 iterations of Apache's *ab* benchmark utility over an 1Gb/s network link. The y-axis displays the average throughput in requests per second as reported by *ab*, and the error bars represent the standard deviation. We performed the experiments requesting files of size 16K, 32K, 64K and 128K from the web server (as listed in the x-axis), and we repeated each test with the corresponding server running: natively, over Pin, with the traditional RPs, and with CRPs. Apache performs on average 8.46% slower when using CRPs, and Pin seems to be responsible for the biggest part of this overhead. Note that this difference drops as the size of the requested file increases. This is due to the workload becoming more I/O intensive (*i.e.,* more data need to be transferred per request) and the number of requests arriving at the server shrinks.

6. RELATED WORK

6.1 Software Self-healing

Software self-healing using RPs was first proposed in ASSURE [30]. RPs are automatically identified and selected by using kernel-level checkpoint-restart, powered by Zap [20] and input fuzzing. RPs were deployed using a modified OS

featuring the Zap virtual execution environment. In contrast to our approach, they require modifications to the OS for deploying RPs and system calls are ignored. Nonetheless, the RP identification component of ASSURE can be used in combination with our work.

Selective transactional emulation (STEM) [31] is a speculative recovery technique that also identifies the function where an error occurs, and it could also be used to assist in identifying RPs. STEM requires source code to analyze errors, and does not support multithreaded applications. Similarly, failure-oblivious computing [29] is another technique that uses a modified compiler to inject code to detect invalid memory writes and correct them by virtually extending the target buffer. This approach is more robust against buffer overflow errors, but comes at significant performance overhead, ranging from 80% up to 500% for a variety of different applications. Moreover, it requires recompilation of the target applications, and it does not handle failures due to other bugs, such as null pointer dereferences.

Instead of attempting recovery, rebooting techniques [34, 13, 9] focus on restoring a system to a clean state. Program restart is a significantly lengthier process than recovery, resulting in substantial application down-time, while data loss is also more frequent. Micro-rebooting aims to accelerate rebooting by only restarting parts of the system, but requires a complete rewrite of applications to compartmentalize failures. These techniques cannot recover from deterministic bugs, and restart all execution threads of a given application. Checkpoint-restart techniques [6, 17] are used in a similar way to rebooting, but restart from a checkpoint. While down time is reduced, they still do not handle deterministic bugs, or bugs maliciously triggered by an attacker (*e.g.,* a DoS attack).

Checkpoint-restart has been also combined with running multiple versions of programs [6]. This approach is based on the assumption that not all versions will be prone to the same error, and it introduces prohibitive costs for most applications, as multiple versions need to be maintained and run concurrently.

Other works have focused on reducing the time from bug discovery to patch generation by automatically generating and applying patches [24, 19, 35]. Unfortunately, automatically applying patches is not very practical, due to the possibility that additional errors are introduced during the patching, or that the patch alters program behavior.

6.2 Coordinated Checkpointing

Our work is also loosely related with work in the area of coordinated checkpointing for distributed systems. Bhargave *et al.* [5] present a checkpoint algorithm for distributed systems, where each process takes checkpoints independently. To recover, a two-phase rollback algorithm is invoked to determine the processes that need to rollback, and the checkpoint they need to rollback to. This is an optimistic algorithm in the sense that it performs well, when errors are infrequent. Independent checkpoint algorithms have the benefit that no coordination between the members of a distributed system is required, but may suffer from the "*domino effect*' [36]'. The "domino effect" occurs when two or more members of the system keep rolling back to previously taken checkpoints in an attempt to reach a globally consistent state, leading to unnecessary delays in the completion of an action.

Sistla *et al.* [32] propose various algorithms based on the asynchronous message logging of incoming messages from individual members of a distributed system. Similarly to our approach, they piggyback tags in the exchanged messages, which are later used to determine the point to roll back and the messages that need to be replayed. In our proposal, we only interleave signaling data in the communications, while we also utilize existing OOB signaling mechanisms provided by TCP and the OS.

7. CONCLUSION

We introduced cascading rescue points, a new mechanism for performing software self-healing on multitier architectures. Our approach enables communicating applications to checkpoint in a loosely coordinated way, so that recovery does not lead to inconsistent states between applications. We intercept existing connections and encapsulate application data using our CRP protocol, which we use to notify remote peers to rollback. We also exploit TCP's OOB signaling to quickly signal peers to stop checkpointing when no faults occur. We implemented a prototype tool that can apply CRPs on binary-only software and evaluate it using the Apache and MySQL servers. We show that it successfully allows them to recover from otherwise fatal errors. In the applications tested, the performance overhead introduced by our approach ranges between 4.54% and 71.96%.

8. REFERENCES

[1] H. Agrawal, R. A. Demillo, and E. H. Spafford. Debugging with dynamic slicing and backtracking. *Software Practice and Experience*, 23:589–616, 1993.

[2] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *Proc. of the Symposium on Security and Privacy*, pages 263–277, May 2008.

[3] A. Arora, R. Krishnan, R. Telang, and Y. Yang. An empirical analysis of software vendors' patch release behavior: Impact of vulnerability disclosure. *Information Systems Research*, 21(1):115–132, 2010.

[4] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53:66–75, February 2010.

[5] B. Bhargava and S. Lian. Independent checkpointing and concurrent rollback for recovery in distributed systems-an optimistic approach. In *Proc. of the* 7^{th} *Symposium on Reliable Distributed Systems*, pages 3–12, October 1998.

[6] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. In *Proc. of the* 15^{th} *ACM symposium on Operating systems principles (SOSP)*, pages 1–11, 1995.

[7] B. Buck and J. K. Hollingsworth. An api for runtime code patching. *Int. J. High Perform. Comput. Appl.*, 14:317–329, November 2000.

[8] C. Cadar, D. Dunbar, and D. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. of the* 8^{th} *OSDI*, pages 209–224, 2008.

[9] G. Candea and A. Fox. Crash-only software. In *Proc. of the* 9^{th} *Workshop on Hot Topics in Operating*

Systems (HotOS IX), May 2003.

[10] J. Etoh. GCC extension for protecting applications from stack-smashing attacks. `http://www.trl.ibm.com/projects/security/ssp/`.

[11] M. Hicks and S. Nettles. Dynamic software updating. *ACM Trans. Program. Lang. Syst.*, 27:1049–1096, November 2005.

[12] M. Howard. A look inside the security development lifecycle at microsoft. MSDN Magazine – `http://msdn.microsoft.com/en-us/magazine/cc163705.aspx`, November 2005.

[13] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton. Software rejuvenation: Analysis, module and applications. In *Proc. of the* 25^{th} *International Symposium on Fault-Tolerant Computing (FTCS)*, page 381, 1995.

[14] InformationWeek. Windows home server bug could lead to data loss. `http://informationweek.com/news/205205974`, December 2007.

[15] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis. libdft: Practical dynamic data flow tracking for commodity systems. In *Proc. of the* 8^{th} *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, March 2012.

[16] A. D. Keromytis. Characterizing self-healing software systems. In *Proc. of the* 4^{th} *MMM-ACNS*, September 2007.

[17] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proc. of the USENIX Annual Technical Conference*, 2005.

[18] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. of the 2005 PLDI*, pages 190–200, June 2005.

[19] K. Makris and K. D. Ryu. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In *Proc. of the* 2^{nd} *EuroSys*, pages 327–340, March 2007.

[20] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of Zap: a system for migrating computing environments. In *Proc. of the* 5^{th} *OSDI*, pages 361–376, December 2002.

[21] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proceedings of the* 33^{rd} *IEEE Symposium on Security & Privacy (S&P)*, 2012.

[22] PaX Project. Address space layout randomization, Mar 2003. `http://pageexec.virtualave.net/docs/aslr.txt`.

[23] PCWorld. Amazon EC2 outage shows risks of cloud. `http://www.pcworld.com/businesscenter/article/226199/amazon_ec2_outage_shows_risks_of_cloud.html`, April 2011.

[24] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *Proc. of the ACM SIGOPS* 22^{nd} *symposium on Operating systems principles*, pages 87–102, 2009.

[25] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: transparent checkpointing under unix. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON'95, pages 18–18, Berkeley, CA, USA, 1995. USENIX Association.

[26] P. Porras, H. Saidi, and V. Yegneswaran. Conficker C analysis. Technical report, SRI International, 2009.

[27] G. Portokalidis and A. D. Keromytis. Fast and practical instruction-set randomization for commodity systems. In *Proc. of the 2010 Annual Computer Security Applications Conference (ACSAC)*, December 2010.

[28] G. Portokalidis and A. D. Keromytis. REASSURE: A self-contained mechanism for healing software using rescue points. In *Proc. of the* 6^{th} *International Workshop in Security (IWSEC)*, pages 16–32, November 2011.

[29] M. Rinard, C. Cadar, D. Dumitran, D. Roy, T. Leu, and J. W Beebee. Enhancing server availability and security through failure-oblivious computing. In *Proc. of the* 6^{th} *OSDI*, December 2004.

[30] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis. ASSURE: automatic software self-healing using rescue points. In *Proc. of the* 14^{th} *ASPLOS*, pages 37–48, 2009.

[31] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. Building a reactive immune system for software services. In *Proc. of the 2005 USENIX ATC*, April 2005.

[32] A. P. Sistla and J. L. Welch. Efficient distributed recovery using message logging. In *Proc. of the* 8^{th} *annual ACM Symposium on Principles of distributed computing (PODC)*, pages 223–238, 1989.

[33] W. R. Stevens, B. Fenner, and A. M. Rudoff. Chapter 24. Out-of-Band Data. In *UNIX Network Programming Volume 1, Third Edition: The Sockets Networking API*. Addison Wesley, 2003.

[34] M. Sullivan and R. Chillarege. Software defects and their impact on system availability - A study of field failures in operating systems. In *Digest of Papers.,* 21^{st} *International Symposium on Fault Tolerant Computing (FTCS-21)*, pages 2–9, 1991.

[35] M. Susskraut and C. Fetzer. Automatically finding and patching bad error handling. In *Proc. of the Sixth European Dependable Computing Conference*, pages 13–22, 2006.

[36] K. Venkatesh, T. Radhakrishnan, and H. Li. Optimal checkpointing and local recording for domino-free rollback recovery. *Inf. Process. Lett.*, 25:295–304, July 1987.

[37] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller. How long will it take to fix this bug? In *Proc. of the* 4^{th} *International Workshop on Mining Software Repositories (MSR)*, 2007.

Twitter Games: How Successful Spammers Pick Targets

Vasumathi Sridharan, Vaibhav Shankar, Minaxi Gupta
School of Informatics and Computing, Indiana University
{vsridhar, vshankar, minaxi}@cs.indiana.edu *

ABSTRACT

Online social networks, such as Twitter, have soared in popularity and in turn have become attractive targets of spam. In fact, spammers have evolved their strategies to stay ahead of Twitter's anti-spam measures in this short period of time. In this paper, we investigate the strategies Twitter spammers employ to reach relevant target audiences. Due to their targeted approaches to send spam, we see evidence of a large number of the spam accounts forming relationships with other Twitter users, thereby becoming deeply embedded in the social network.

We analyze nearly 20 million tweets from about 7 million Twitter accounts over a period of five days. We identify a set of 14,230 spam accounts that manage to live longer than the other 73% of other spam accounts in our data set. We characterize their behavior, types of tweets they use, and how they target their audience. We find that though spam campaigns changed little from a recent work by Thomas et al., spammer strategies evolved much in the same short time span, causing us to sometimes find contradictory spammer behavior from what was noted in Thomas et al.'s work. Specifically, we identify four major strategies used by 2/3rd of the spammers in our data. The most popular of these was one where spammers targeted their own followers. The availability of various kinds of services that help garner followers only increases the popularity of this strategy. The evolution in spammer strategies we observed in our work suggests that studies like ours should be undertaken frequently to keep up with spammer evolution.

Categories and Subject Descriptors

K.4.1 [**Computers and Society**]: Public Policy Issues - *Abuse and Crime Involving Computers*

General Terms

Security, Measurement

Keywords

Spam, Twitter, Online social networks (OSNs)

*Sridharan and Shankar participated in this work when they were graduate students at Indiana University.

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

1. INTRODUCTION

Email spam has been a problem for decades. As email spam filtering programs have improved, with many claiming 99% or higher accuracies, spammers have looked for other avenues. Online social networks (OSNs), such as Twitter, are obvious new targets because of their sharp rise in popularity. Twitter alone boasted 140 million users as of March 2012 [20]. Fighting spam on OSNs requires new types of filtering techniques, owing to the fundamental differences between email and OSNs as communication media. As an example, in the case of Twitter, the unit of communication, *tweet*, is merely 140 characters, limiting the efficiency of traditional spam filters which rely on a certain size of text to be effective. Owing to the differences, both operators of OSNs as well as the research community have recently been actively pursuing the topic of spam on OSNs.

An over-arching theme across recent spam-related research work on Twitter has been the characterization of spam activity and building classifiers to aid in the identification of spammers and spam tweets. Our work complements existing research around this theme by examining an untouched aspect of Twitter spam, which is that we do not know how spammers pick their targets. Having this knowledge can complement existing efforts to defeat spam by protecting the targets and also by identifying spammers. Interestingly, this investigation is possible on OSNs because they are contained ecosystems where both spammers and the targets being spammed reside as users within the same system. This is in sharp contrast to traditional email spam where email addresses sending and receiving spam belong to different administrative entities.

In our work, we examine over 80K spam accounts on Twitter and investigate their targets. The strategies we bin spammers under are motivated by a combination of the functionality provided by popular softwares known to automate spam on Twitter and a study of the behavior of worst offenders. The key contributions of our work are along the following two dimensions:

- **Strategies for picking targets:** We find five key strategies being used by Twitter spammers. We find that

Twitter spammers overwhelmingly target their own followers, who are expected to be voluntary subscribers of spammers' tweets. This finding is in sharp contrast to the observation made by Thomas et al. [16] just about a year ago where they found that spammers often failed to garner followers. A plausible explanation for the difference in observation is that spammers have evolved to adapt around Twitter's effort to fight spam [19]. This view is supported by observations about the thriving underground economy around buying Twitter followers [13], which makes it easy for spammers to adapt in the manner we observed.

In terms of other strategies used to find targets, we find that a relatively smaller number of spammers also target the followers of other popular accounts and even search for targets whose tweets contain keywords of interest to their spam campaigns. Finally, an even smaller number of spammers hijack popular discussion topics (referred to as *trending topics*, in Twitter parlance) to increase the possibility that their tweets are found by a large number of users interested in following those topics. The knowledge about these strategies can be used to actively identify spammers and to protect their targets from being spammed.

- **Observations about spammer behavior:** During the course of our work, we made multiple observations about spammer behavior that can be used as features to improve the performance of existing spam classifiers for Twitter. First, almost 3/4th of the spammers use a certain tweet type exclusively when only 13% of good users on Twitter have this behavior. In fact, 2/3rd of the spammers target only their own followers through the spam tweets. In contrast, only 10% of good Twitter users target only their own followers. We also learnt that the method used for posting tweets can also predict whether the tweet is spam or not, primarily because spammers seem to prefer different modes of posting their tweets than other Twitter users.

The key motivation for our work was to complement existing spam defenses and we believe that the insights gained can effectively serve that purpose. Additionally, we were able to draw comparisons with Thomas et al.'s work done a year ago, where they examined the behavior of Twitter spammers. The comparisons revealed that even though the spam campaigns have remained the same, Twitter spammers have evolved in their strategies in a mere one-year period in an attempt to stay afloat amidst Twitter's anti-spam efforts. This observation suggests that strategies of spammers need to be re-examined frequently in order to stay ahead of the game.

The rest of this paper is organized as follows. In Section 2, we discuss the methodology for our data collection and how we differentiate successful from unsuccessful spammers in our data set. Section 3 provides background information about types of tweets on Twitter and discusses our observations on the types used by spam profiles. In Section 4, we describe the major strategies we found which are used by spammers to find their target users. In Section 5, we provide our observations regarding applications used by spammers. Section 6 provides a discussion on some interesting observations in our data set. Sections 7 and 8 provide related work and conclusions, respectively.

2. DATA COLLECTION AND OVERVIEW

We used Twitter's streaming API to collect tweets for November 1st, 2011. The API samples one in ten tweets and makes them available to us. Our data set contained 19,991,050 tweets and 7,078,643 unique Twitter user account profiles. Since we needed a set of spam profiles to test, we used the output of Twitter's suspension policy as ground truth. To determine which accounts were suspended by Twitter, we visited `http://www.twitter.com/<username>` for each username contained in the profiles in our data set and looked for a string indicating suspension of the account. This process yielded 82,274 suspended profiles. These profiles are similar in nature to those analyzed by Thomas et al. in there recent work where they analyzed suspended Twitter accounts. While we analyze this data for comparison and to see how Twitter spammers have evolved in the last one year since Thomas et al.'s work, we further applied a couple of filtering criteria in order to analyze spammer strategies. First, we eliminated profiles that tweeted in languages other than English since otherwise we would not be able to analyze their tweets against various spamming strategies. This reduced our dataset to 53,083 profiles. Further, to run each profile against various spamming strategies, we needed a certain number of tweets from each. We set a threshold of 10 tweets within five days of the data collection day as our threshold and eliminated profiles that tweeted less number of tweets within the 5-day period. The choice of a 5-day period was guided by the intuition that a profile taking much longer than 5 days to tweet ten times is either inactive during our data collection period or was suspended for reasons other the sending high volume spam. This step pruned our data set to 14,230 profiles. We refer to these profiles as *successful spam profiles* subsequently in this paper and the rest of the suspended profiles as *unsuccessful spam profiles*.

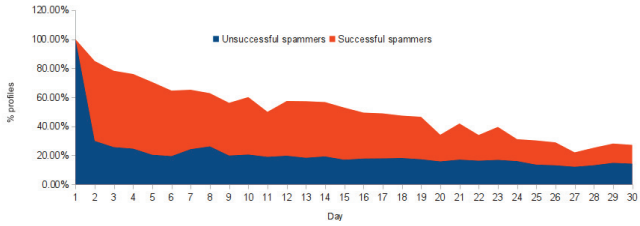


Figure 1: Lifetime of successful and unsuccessful spam profiles over one month

In the 5-day period, the entire group of 82,274 suspended profiles tweeted 1,353,340 times, of which 2/3rd of the tweets were from successful spam profiles, which comprised only 17% of the profiles. In fact, almost half of the tweets from unsuccessful spam profiles arrived on the first day of our data collection. These numbers indicate that most of the activity from unsuccessful spam profiles is front loaded, perhaps because they get suspended quickly. Figure 1 shows the percentage of successful and unsuccessful spam profiles that survive on each day for the month of November, 2011. Since we do not know how long any profile was alive before November 1st or how long it lived after November 30th, the lifetimes in this Figure are lower bounds, in that we underestimate the lifetime of all profiles. This Figure shows

Type	Successful spammers		Unsuccessful spammers		Other users	
	% Tweets	% Profiles	% Tweets	% Profiles	% Tweets	% Profiles
Regular	89.1%	91.2%	54.5%	78.7%	41%	93%
Replies	4.7%	20.9%	21.6%	32.8%	37%	80%
Mentions	5.1%	20.1%	18.4%	24.9%	11%	50%
Retweets	1.1%	8.6%	5.5%	9.5%	8%	49%
Total tweets	917,258		436,047		1,919	
Total profiles	14,230		68,044		100	

Table 1: Tweet types of successful and unsuccessful spam profiles and of regular Twitter users

that 70% of unsuccessful spam profiles and 15% of successful spam profiles get suspended on the first day. Work done by Thomas et al. a year ago [16] looked at all spammers' collectively and found that 77% of spam profiles were suspended on the first day and 92% within three days. We suspect that since unsuccessful spam profiles are a larger fraction of profiles in our data, that they dominated their data set as well, potentially masking the behavior of successful spam profiles.

3. TWEET TYPES

Twitter users can send various types of tweets. A basic tweet, referred to as a *regular* tweet in the rest of this paper, is received by all followers of a sender. It appears both on sender's *home timeline* as well as the *home timeline* of each of the sender's followers.

Three other kinds of tweets are allowed in Twitter. The first is called a *reply* tweet, which is sent as a reply to a tweet. In addition to appearing on the *home timeline* of the receiver provided they are following the sender, a reply tweet appears on the *home timeline* of anyone following both the sender and the recipient. The second type of tweet is a *mention* tweet, which is a tweet directed at a specific Twitter user. It is much like a reply tweet except that it is more broadly visible since it appears on the *home timeline* of anyone who is following just the sender. Finally, the last type of tweet is a *retweet*, which is a mechanism to forward tweets to one's followers. This is visible on the *home timeline* of each of the sender's followers.

Note that each user's *home timeline* is private to the user, implying that the tweets appearing there cannot be seen by anyone else. In turn, this implies that no one, including spammers, can misuse any information contained in the tweets, such as recipients or tweet content. However, tweets sent by each user also appear on his/her *profile timeline*, which is public by default. Tweets contained there can be searched and exploited by spammers.

Table 1 shows the tweet types seen in our data. In order to compare spammer behavior with that of other users, we randomly picked 100 regular profiles in our data. We see that an overwhelming number of profiles across both spam and other users used regular tweets. *Successful spam profiles made the heaviest use of regular tweets, with 89% of their tweets being regular.* In comparison, half or less of the tweets from unsuccessful spam profiles and other users were regular. Reply tweets, followed by mentions, were also popular among unsuccessful spam profiles as well as regular users. Successful spam profiles made relatively less use of these types of tweets, perhaps because Twitter is known to suspend accounts which send large numbers of replies or mentions [19].

Retweets were the least popular for both types of spam profiles, with successful ones using them even less than the unsuccessful ones. Even though retweets were the least popular among other users as well, they still commanded a significant fraction of their tweets. This finding is intuitive since spammers get very little leverage out of retweets, which are similar to unedited forwards of email messages but without any control over the target audience since retweets only go to one's followers.

We note that work done by Thomas et al. a year ago [16] found that 52% of spam profiles made use of mention tweets. The corresponding fractions for successful and unsuccessful spam profiles in our data are 1/5th and 1/4th respectively. For other users, it is 50%. Since their data set was sampled similar to ours, we conclude that Twitter spammers have evolved their strategies in the last one year. Finally, when looking at what percentage of profiles made use of a specific type of tweet, Table 1 suggests that spammers are using fewer types of tweets compared to other users overall. We investigate this issue in detail next.

Table 2 shows the percentage of profiles that exclusively used one type of tweet. We find that over *3/4th of successful spam profiles exclusively used only one type of tweet when the corresponding percentage for unsuccessful spam profiles is 2/3rd and for other users is 14%.* This observation suggests that spammers focus on a limited set of spamming strategies while regular users use various tweet combinations. Further, 2/3rd of successful spam profiles exclusively use regular tweets, when the corresponding percentage for unsuccessful spam profiles is about half and that for good profiles is 11% (see Table 2). Both of these characteristics could be used as features in identifying spam profiles.

Tweet type	Successful spammers	Unsuccessful spammers	Other users
Exclusively regular	68.3%	49.3%	11%
Exclusively replies	4.5%	6.8%	2%
Exclusively mentions	2.4%	10.5%	1%
Exclusively retweets	0.28%	1.0%	0%

Table 2: Spammer profiles using a single type of tweet exclusively

4. STRATEGIES FOR PICKING TARGETS

In the following sections, we describe the different strategies employed by successful spammers in finding their targets.

4.1 Spamming Ones Own Followers

We saw in Section 2 that over 2/3rd of successful spam profiles exclusively use regular tweets to send spam. Clearly,

```
http://www.amazon.com/Duke-Blue-Devils-Foot-Bean/dp/B000UIKZKC?SubscriptionId=AKIAJ7TIJCJQNX6EL3PQ&tag=
scot0e-20&linkCode=sp1&camp=2025&creative=165953&creativeASIN=B000UIKZKC&utm_source=twitterfeed&utm_
medium=twitter
```

Figure 2: An example Amazon affiliate program link (affiliate ID=scot0e-20)

for their spam campaigns to succeed, spam profiles targeting only their own followers need to have followers. (A discussion of the methods spammers employ to find their followers is in Section 6.2.)

We begin by examining the followers of all spam profiles present on the first day of our data collection. Figure 3 shows the number of followers for both successful and unsuccessful spam profiles. We note that about 1/3rd of successful spam profiles have over a 100 followers. In contrast, only 1/6th of unsuccessful spam profiles reach that number. On the other hand, only 5% of successful spam profiles have zero followers and about 1/3rd have less than 10. In comparison, nearly 40% of unsuccessful spam profiles have zero followers and a total of 2/3rd have less than 10. Thomas et al. noted in their work that 89% of spam profiles have less than 10 followers. The contrast with our observations makes us believe that not only have all spammers become smarter about acquiring followers but also that the successful spammers fare better in their follower counts. Focusing on the follower count of the group of 2/3rd successful spam profiles that exclusively used regular tweets (Figure 3), we note that the *spam profiles that use exclusively regular tweets maintain a large number of followers, with almost 1/3rd of them having 100 or more followers.*

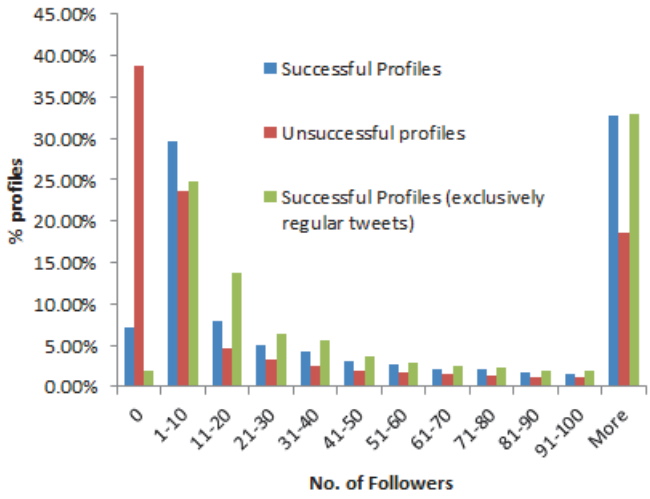


Figure 3: Follower counts

4.1.1 Spam Campaigns

To gain insights into spam campaigns of spammers using this strategy, we examine the regular tweets of all the 14,230 successful spam profiles. We only consider ones for which we have at least 10 regular tweets with links in our data. The cut-off ensures that we have sufficient tweets to judge the nature of their campaigns. This cut off left us with 7,704 spam profiles. In order to study the destinations of their spam campaigns, we require that 80% of the final destination links (upon traversing all HTTP redirects for each URL) for each profile lead to the same domain name. This heuristic ensures that we capture where these links lead while allowing for host-name and directory-level variations in URLs, which spammers often leverage to create distinct-looking links.

A total of 6,630 (86% of those with 10 links in our data) spam profiles met our criterion and were leading their followers to 559 different domains. Two of the domains were particularly noteworthy. In the first, the final destination was `t.co`, which is Twitter's own URL shortener. Since URL shorteners are never the destination page, we looked up the destination pages manually and found that they were Twitter's warning pages, informing the visitor that the link in the tweet was leading a malware-serving site. (The user still has the option of clicking on the original destination.) A total of 1,822 profiles led to a `t.co` warning page and clearly contained one or more spam campaigns leading to malware.

The second interesting domain was where 1,741 profiles were acting as *Amazon affiliates* because they were each leading their targets to `Amazon.com` URLs similar to the one shown in Figure 2. Thomas et al.'s work also found a large number of affiliate spam. However, the spam tweets in their data used mention tweets, while we find spam profiles to be using regular tweets instead, indicating a shift in strategy. Almost all of the spam URLs had `amzn.to` as the starting point landing page in tweets, which is `bit.ly`'s specialized URL shortening service for Amazon. `Amazon.com` accounted for over 55% of the 917,294 tweets by successful spam profiles, and contained a total of 76 unique Amazon affiliate IDs.

In order to understand the different `Amazon.com` campaigns, we examined the affiliate IDs and the spam profiles associated with them. All profiles using the same affiliate ID were clearly part of the same campaign. Common profiles across multiple affiliate IDs also implied that they belonged to a spam campaign. We found 19 `Amazon.com` campaigns in our data set where the spammers were using regular tweets to target their followers. Table 3 shows the top five campaigns where a spammer either used a large number of Twitter profiles or affiliate IDs. The largest of these involved over 1.5K Twitter profiles and sent almost half a million tweets over our data collection period.

Amazon campaign	Twitter profiles	Affiliate IDs	Tweets
1	1,519	55	446,552
2	163	1	46,848
3	18	1	4,853
4	8	4	4,441
5	3	1	43
Unique spam profiles across all 19 campaigns		1741	

Table 3: Top-5 Amazon affiliate spam campaigns where spammers used regular tweets to target their own followers

4.1.2 Spam Profiles Binned

Of the 14,230 successful spam profiles, 92.1% (12,979) used regular tweets and 68.3% (9,723) used regular tweets exclusively. The analysis in this section suggests that 6,630 spam profiles were targeting their own followers in their spam campaigns. It is noteworthy that of these, 93.6% (6,208) were using regular tweets exclusively.

4.2 Spamming Followers of Popular Profiles

As an alternative to targeting one's own followers, spammers can find suitable targets by simply targeting other profiles' followers. This is advantageous in cases such as when, say, a spammer wants to target music lovers. He/she can send spam to the followers of music celebrities, leveraging the fact that many of the celebrities have already acquired a large number of followers. Targeting the followers of popular profiles is in general an easy strategy a spammer can use, even if the goal is not to target specific type of users. In fact, softwares such as TweetAttacks [18] and TweetAdder [17] have readily available automation spammers could exploit to target followers of any profile. Note that the tweets sent using this strategy will have to make use of reply or mention tweets since otherwise, they will only go to the profile's followers.

A total of 4,086 (28.7%) spam profiles made use of reply or mention tweets in our data. In order to determine how many of them targeted followers of other profiles, we needed to have at least a few samples of the users who received their spam. We used a threshold of four unique Twitter users receiving spam from a specific spam profile and checked if at least 50% of the targets for each spammer were following the same profile. If so, the spammer likely targeted the followers of that profile.

Of the spammers that made use of reply or mention tweets, 3,528 (86.3%) had targeted at least four different users in their spam. For each spammer's targets, we collected information about all profiles each target was following using the Twitter API. However, since API calls are expensive, we collected only the first 5000 followings for each target. When checking if at least 50% of the targets for each were following a specific profile, we found that 877 profiles fit the heuristic. They contributed an average of just under 23 reply or mention tweets each, which is much lower than the average of 64 tweets an average spam profile in our data contributed (see Table 1). This suggests that the volume of spam from this strategy is not high. Examining the tweets of this group of 877 spam profiles, only 272 had more than 10 reply or mention tweets with links. Of these, 225 profiles had more than 80% of their links going to the same domain, implying that these profiles were pointing their targets to a specific domain and potentially running spam campaigns.

The most popular of these domains was `hellojb.info`, which made up 18% (157) of the total 877 profiles and 18% of the total reply or mention tweets from this group of spammers. 149 of the profiles from this campaign used reply tweets to target the followers of celebrity Justin Beiber with the link that claimed to have some information on how to get Justin Beiber to follow them.

The second most interesting domain was where six spam profiles were acting as Amazon affiliates. Their tweets carried four unique affiliate IDs, suggesting that there are up to four spam campaigns here (see Table 4). Though of a much smaller scale, these campaigns are of a similar nature to the Amazon affiliates campaigns found in Section 4.1. However, since none of the profiles or affiliate IDs here are common with those found in Section 4.1, we believe that the spammers behind these campaigns are a different group from those found in Section 4.1.

Amazon campaign	Twitter profiles	Affiliate IDs	Tweets
1	3	1	34
2	1	1	14
3	1	1	11
4	1	1	11
Unique spam profiles across all 4 campaigns		6	

Table 4: Amazon affiliate spam campaigns where spammers used reply or mention tweets to target others' followers

4.2.1 Spam Profiles Binned

A total of 877 spam profiles were binned under this strategy. Of these, 26 were common with the 6,630 binned under Section 4.1. These were clearly targeting their own followers through regular tweets and others' followers through reply or mention tweets. Overall, thus far, we have binned 7,481 (52.5%) of the 14,230 profiles of successful spammers.

4.3 Spamming based on Keywords in Tweets

Spammers can also pick their targets based on the content of tweets from Twitter users. For example, a spammer who has a campaign revolving around phones may target users who have used the term "phone" in their tweets. Finding such targets is easy for spammers since Twitter already facilitates searching tweets based on keywords. In fact, that softwares such as TweetAdder already support automating this spamming strategy suggests that some spammers may be making use of it.

Spammers using this strategy could use reply tweets or mention tweets to send spam to their chosen targets. However, looking at a mention tweet, one cannot determine which specific tweet triggered a response from the spammer. This rules out the scrutiny of 1,408 spam profiles that used mention tweets but not reply tweets. Fortunately, it is possible to check for the use of this strategy when reply tweets are used, since reply tweets contain an identifier, *status ID*, for the original tweets that were replied to.

2,969 successful spammers in our data (21% of 14,230) made use of reply tweets, some in conjunction with other types of tweets. In order to judge if a spam tweet contained a keyword, we first had to identify possible keywords. Towards this goal, we used the term frequency-inverse document frequency (TF-IDF [8]) statistic which reflects how important a word is to a document in a collection. Upon computing the TF-IDF score of about 7 million words present in the tweets from successful spammers, we picked the top 50K as possible keywords. Then for each reply tweet of each spam profile that had at least three reply tweets, we extracted the source tweets (tweets they were replied to) by using the status ID present in the tweets and querying the Twitter API. In the next step, we looked for common keywords in source tweets for the each spam profile. If a word appeared in over a percentage threshold of a spam profile's tweets and was one of the 50K words we chose as keywords, we took it to imply that the spammer targeted authors of tweets with

specific keywords. Since we had varying number of source tweets for each spammer, we used four slabs to judge if a spam profile was using this strategy to find targets. For the cases where we had exactly 3 tweets, it was important to set a very high threshold. So, we set a threshold of 100%, implying all 3 source tweets had to have the keyword in question. For profiles with more than three but less than 10 tweets, we required that at least 50% of the tweets had to have the keyword. Further, for cases when we had more than 10 but less than 20 tweets, we relaxed the threshold to 40%. Finally, when more than 20 tweets were available, we required that at least 30% should have the keyword.

Of the 2,969 spam profiles that used reply tweets, 2,419 (81.4%) had at least three reply tweets each. Of these, 1,004 (41.5%) passed the heuristic to fit this strategy. Further, 710 of these profiles had at least 10 tweets with links and at least 80% of their links were going to a single domain, implying that these spam profiles were likely running campaigns. The most popular domain was `jbet.info`, which involved 173 profiles that were promoting this health-related website using keywords "breakfast", "lunch", and "dinner" to locate targets. The next most popular, `hellojb.info`, was also caught in Section 4.2 since the spammer was targeting the followers of celebrity Justin Bieber. It was caught here due to the presence of keyword "justinbieber" in the tweets of targets and the profiles involved in both cases were the same set. Finally, the Amazon affiliate marketing campaign featured prominently in this category as well. We saw six different campaigns using a diverse range of keywords, ranging from "buy", "iPhone", "weight", "aging", and "jailbreak" (see Table 5). The largest two campaigns involving 73 and 19 spam profiles respectively, had three and one profiles common with Amazon campaigns found in Section 4.2. Also, campaign 4, involving three spam profiles, shared a profile with Section 4.2. A closer inspection of the common profiles suggests that the spammers simply targeted keywords and got followers of specific popular profiles as a result.

Amazon campaign	Twitter profiles	Affiliate IDs	Tweets
1	73	1	1,059
2	19	1	245
3	5	2	66
4	3	1	30
5	1	1	15
6	1	1	13
Unique profiles across all 6 campaigns		102	

Table 5: Amazon affiliate spam campaigns where spammers used keywords to find targets

4.3.1 Spam Profiles Binned

A total of 1,004 spam profiles were binned under this strategy. Of these, only one was found in Section 4.1 while 210 profiles were found in Section 4.2. Of the latter, 150 were simply due to the `hellojb.info` campaign discussed earlier. Overall, thus far, we have binned 8,274 (58.1%) of the 14,230 profiles of successful spammers.

4.4 Trending Topics Hijacking

Twitter users can tag their tweets with a topic by using a keyword after the symbol '#'. This keyword is referred to as a hashtag in Twitter parlance. Popular hashtags become trending topics, which attract visibility. Spammers have been known to hijack trending topics to increase the visibility of their spam campaigns [16]. Here, we examine how many of the successful spammers in our data set are exploiting this strategy. Note that spammers can make use of this strategy in conjunction with other spamming strategies, such as spamming their own followers or followers of popular profiles.

A spammer using this strategy may use any of the various types of tweets, regular, mentions, or replies so long as the tweet contains a hashtag. We analyzed the 4,327 spam profiles in our data set that had at least one hashtag in their tweets. In order to identify spammers who tried to hijack trending topics, we pre-calculated a set of 200 most popular hashtags out of the total 466,597 hashtags present in the tweets in our data set. Next, just like in Section 4.3, we required that each spammer profile have a minimum of three tweets with hashtags. This ruled out 824 profiles leaving us with 3,503 (81%) profiles to examine. On each profile, we applied the same slabs as we applied to determine keywords in Section 4.3 to determine whether or not it abused a popular hashtag for spam distribution.

Our heuristic concluded that 1,043 spam profiles hijacked trending topics. Of these, 174 likely ran a campaign since they led their targets to a specific domain as they had at least 10 hashtag tweets with links and 80% of their tweets pointed to a particular domain. The top campaigns used fewer profiles and proportionally fewer tweets compared to those found in Section 4.3 and were promoting politically-oriented, gaming-related and various kinds of unsavory websites. Two domains in these campaigns were the most interesting. The first was the Amazon affiliate ID campaign, which was found in all previous sections. Here, we found five spam campaigns (shown in Table 6). All but one campaign were already identified in Section 4.1, indicating that spammers simply tried to cast their net wide. The second interesting domain was `adf.ly`, where 19 spam profiles were hijacking trending topics to advertise this ad-based shortener. Such shorteners were observed by Thomas et al. also. The revenue model for these shorteners is to show advertisements before leading the visitor to the destination link.

Amazon campaign	Twitter profiles	Affiliate IDs	Tweets
1	5	5	836
2	1	1	471
3	1	1	40
4	1	1	31
5	1	1	10
Unique profiles across all 5 campaigns		9	

Table 6: Amazon affiliate spam campaigns where spammers hijacked trending topics to gain attention

4.4.1 Spam Profiles Binned

Given that this strategy merely requires the addition of an extra hashtag to a spam tweet, it can be easily combined with other strategies. Hence, we see significant commonality between profiles binned under this strategy and others discussed before. Specifically, we found that 523 of the 1,044 profiles binned under this strategy were using regular tweets to spam their followers (discussed in 4.1). Another 14 profiles were common with Section 4.2. Finally, three profiles

were common with those in Section 4.3. Overall, this strategy binned 503 new spam profiles not binned through previous strategies, bringing the total binned profiles to 8,777 (61.7%) of the 14,230 profiles of successful spammers.

4.5 Targeting Own Followers by Retweets

We saw in Section 4.1 that regular tweets are a popular way for spammers to target their followers. Retweets can be used as an alternative because they also reach ones followers, except that they are restrictive because the tweet content has to be borrowed verbatim from another tweet and cannot be modified. However, according to the TweetAttacks software, retweets have a higher click rate than normal tweets, which may make spammers prefer them. In fact, TweetAttacks referred to the strategy of using retweets as "retweet attacks".

Retweets were the least popular among spammers as well as other users, with successful spammers using them even lesser than others (see Table 1). 1,230 successful spam profiles used retweets at all. Of these, only 70 had greater than 10 retweets with links and only 28 had 80% of their links pointing to a single domain, suggesting that they were running campaigns. Incidentally, each of these 28 were using retweets exclusively. 26 of them were retweeting content from a Twitter profile called *omgwire*. The profile seemed to be promoting a celebrity gossip website, `omgwire.com`, so the retweets were tailored to that promotion. The other two domains were also retweeting content from two separate Twitter profiles and promoting a website each.

4.5.1 Spam Profiles Binned

This strategy only helped bin 28 additional spam profiles but none were common with those found under previous strategies. Overall, we have thus far binned 8,805 (61.9%) spam profiles of the total 14,230 profiles of successful spammers.

5. POSTING METHODOLOGY

Figure 4: Examining the "via" field at the bottom of a tweet helps determine how it was posted

Twitter allows posting tweets using a variety of methodologies, including their Web interface and various Twitter-provided and third party clients which post through the Twitter API. Examples of such clients include Tweet Button, Twitter's mobile interface, and specialized applications for various smartphone platforms, such as BlackBerry, iPhone, and Android. Additionally, there are RSS-to-tweet services such as *twitterfeed*,`dlvr.it` and Google's client for the users of their blog service,blogspot.com that automatically post a link to Twitter account each time a new content is published on a blog.

Twitter facilitates knowing how a tweet was posted. For example, the bottom of the tweet shown in Figure 4, which is one of the spam tweets from the Amazon campaign discussed throughout Section 4, shows that it was posted via Twitterfeed, a third-party client.

Looking into how successful and unsuccessful spam profiles differ in their posting behavior versus other users helped draw interesting conclusions which we present next. Table 7 shows the top six most popular ways of posting, based on the number of tweets posted and the number of profiles that make use of it. We find that *Twitterfeed is the most popular among spam profiles, with successful ones exploiting it for 2/3rd of their tweets.* The Web is popular among all three groups, with close to 40% of the profiles in each group posting via the Web. Note that popular software, such as TweetAdder and TweetAttacks, automate their posts so they appear to have been posted via the Twitter Web interface. Twitter's mobile Web interface, which is the version of Twitter.com for mobile devices, is popular among unsuccessful spammers but not among successful spammers and other users.

Also, there is evidence of spammers using fewer dedicated apps when compared to other users as shown in Table 8. The average suggests that the organic profiles use several different apps, where as spammers (successful or unsuccessful) have fewer dedicated apps.These findings indicate that the method of posting tweets itself could be used to predict if a tweet is more likely to be spam versus good.

Type of profile	No. of apps used	users per app
Other users	65	0.68
Successful	790	18.01
Unsuccessful	1,599	42.55

Table 8: Average number of apps used by a single profile under each category of users

6. DISCUSSION

In the previous sections, we discussed how successful spam profiles find their targets and what tools are often associated with spam accounts. In this section, we discuss two additional aspects of our study. First, we look at the set of successful spam accounts which were not accounted for in any of the strategies described in Section 4. We discuss what strategies may have been missed and offer explanations based on limitations in our dataset to discover such profiles. Next, we discuss the concept of gathering followers. As was shown in 4.1, a large number of successful spammers have a significant number of followers. We briefly investigate what methods are used by spam accounts to gather followers and remark upon two major types of follower gathering schemes.

Type	Successful spammers		Unsuccessful spammers		Other users	
	% Tweets	% Profiles	% Tweets	% Profiles	% Tweets	% Profiles
twitterfeed	65.3%	25.1%	8.18%	5.4%	3.5%	4.0%
Web	13.3%	38.7%	41.5%	45%	24%	45%
Tweet Button	3.5%	15.8%	2.5%	3.7%	0.0005%	1%
Mobile Web	3.3%	8.3%	10.6%	24.7%	4.5%	9%
dlvr.it	3%	2.7%	1.44%	1.4%	0.46%	1%
Google	1.1%	1.7%	0.2%	0.3%	0%	0%
Total tweets	917,258		436,047		1,919	
Total profiles	14,230		68,044		100	

Table 7: Tweet posting methodology of successful and unsuccessful spam profiles and of regular Twitter users

6.1 Unbinned Spam Profiles

We discussed five strategies that successful spammers use to pick their spam targets in Section 4. Collectively, these strategies helped understand the spamming behavior of 62% of the 14,230 profiles of successful spammers. In trying to understand the behavior of unbinned spam profiles, we learnt a few things. First, of the 5,425 unbinned profiles, the largest chunk, 64.7%, exclusively made use of regular tweets and could not have directly targeted any other Twitter users but their followers. Owing to this observation, they could only be binned under Section 4.1 amongst the strategies we describe. Recall that in Section 4.1, we considered a profile binned if it appeared to be running a campaign, defined by at least 10 tweets with links and 80% of the links leading to a single domain. Since alternative definitions for campaigns is possible, we tried a few different ones to see their impact on the proportion of unbinned profiles. For example, if we require that a profile only have at least five tweets with links and when the number of tweets with links is less than 10, 80% of them be leading to a specific domain and when the number of tweets with links is 10 or more, 50% of them be leading to a specific domain, we bin 8,034 profiles in Section 4.1 as against 6,630. In turn, 72% spam profiles get binned. This simple exercise suggests that further experimentation with definitions of campaign is one avenue for binning more spam profiles.

Of the remaining 1,910 unbinned profiles, 89.7% sent at least one mention tweet and 328 sent exclusively mention tweets. Note that of the two strategies involving mention tweets, described in Sections 4.2 and 4.3, the latter had to ignore mention tweets due to the unavailability of keywords spammers may have targeted from users' original tweets. This caused us to miss an opportunity of binning more spam profiles that used mention tweets.

Further, spam automators, such as TweetAdder, provide other ways spammers could use to pick their targets. For example, targets could be chosen based on their geographical location and language, which are pieces of information typically associated with Twitter profiles. However, given that the profiles we investigated were already suspended, we did not have access to information about their followers. Such information may have helped us investigate more strategies used, especially by regular tweeting profiles, to identify targets.

6.2 Garnering Followers

Given that about 90% of successful spam profiles make use of regular tweets and 2/3rd of them use them exclusively, how spammers garner followers is an interesting consideration. Indeed, Figure 3 confirmed that a large number of successful spammers have a respectable number of followers.

There are many ways spammers can use to garner followers. A popular mechanism is to become a part of one or more peer-driven communities that encourage following back. We discovered two interesting communities in our data, `#InstantFollowBack` (#IFB) and `#TeamFollowBack` (#TFB).

The first of these is controlled by a third-party client under the Twitter profile name 'instantfollowBA' that allows a Twitter user to find and follow users on Twitter by requiring them to be listed under it's follow back community #InstantFollowBackGradeA. The requirements for this is to have public account with minimum 500 followers and follow back 99% of the them on a every day basis which the client app claims to make it easier by providing a few tools. Additionally, it rewards each user with points that offer different levels of promotions. Each user is given a 'status' that indicates the number the 'Gold' and 'Experience' points earned by them. Users can increase their points by being a part of #InstantFollowBackGradeA and by tweeting (advertising) about 'instantfollowBA'. As the level of a user increases, more points are awarded. After collecting a sufficient number of points, a user gains center stage in promotions, such as retweets to all of the main #IFB profile's followers, retweets by others in the community to increase the profile's visibility in Twitter, display of banner ads by several members of the #IFB community and so on. This method of incentivizing users to join a follow back community is unique though its popularity may be limited due to the complicated nature of the setup. In fact, we found only 217 profiles from our data set that were involved in this scheme. The second community in our data, #TFB, is a Twitter hashtag used by numerous follow back groups with goals similar to #IFB. Profiles involved in these schemes sign their tweets with the #TFB hashtag. Often, they add the hashtag to their publicly searchable profile information. This makes it easy for anyone requiring followers to find profiles willing to follow them back without regard to who is requesting or what content is being tweeted. We found 509 profiles in our data using the #TFB hashtag, all in Section 4.4.

Yet another option for increasing the number of followers is to buy them. Owing to the importance of collecting followers, websites such as, `http://getfollowrsontwitter.org/`, have cropped up that allow one to buy followers, often in increments of thousands. At the same time, online marketplaces for services, such as `fiverr.com`, routinely feature services where offerers guarantee thousands of Twitter followers for as little as $5 (see Figure 5 for an example of such an offering). Works in [6, 13] studied Twitter account mar-

kets and confirmed that buying followers is indeed prevalent on Twitter.

Figure 5: An advertisement offering to gather Twitter followers

It is important to note that the techniques described thus far only go as far in finding followers. Specifically, they are unlikely to work well for spam profiles that require relevant followers, such as in the case of exclusive regular tweeters. As described before, these are the largest chunk of successful spammers and require more intelligent strategies for garnering followers. To garner such followers, spammers could use the strategies we described in Sections 4.2 and 4.3 to locate targets and friend them. Additionally, a spammer may use publicly searchable profile information to target people based on location, interests etc. Since a good number of these targets may be genuinely interested in the content promoted by a spammer, it may increase the likelihood of them actually following the spammer back. While we have seen evidence of such activity in a few spam profiles that are alive, it is in general difficult to prove if the followers of a profile were gathered using a certain strategy. In fact, spamming only one's followers thins the line between genuine and spam content since the only real violation by spam accounts tweeting only to their followers is the use of an automated tool to send tweet in large volumes. In contrast, the dominant spammers in Thomas et al.'s work were blatant violators of Twitter terms and conditions since they largely made use of unsolicited mentions to promote spam tweets.

7. RELATED WORK

Spam on online social networks has been analyzed in a number of different ways. Benevenuto et al. [2] analyze online video spam on Youtube and employ machine learning techniques to identify spammers on YouTube. The study by Gao et al. [5] involves detecting and characterizing spam campaigns on Facebook. Here, we focus primarily on works related to Twitter, focusing particularly on previous research which is most related to our work and influenced our investigation.

Much of the research on Twitter spam has focused on building classifiers that distinguish spam profiles from non-spam profiles or spam tweets from non-spam tweets. Lee et al. [10] and Stringhini et al. [14] investigated spam on Twitter using *social honey pots*, which are profiles created specially for the purpose of being spammed and proposed a machine learning based approach to classify profiles that send spam to these accounts. Lee et al [9] also studied collective spam on Twitter by analyzing how cyber criminals exploit *trending topics* to propagate spam and devised a machine learning based methodology to detect them. Yang et al. [22] studied tactics used by spammers and employed machine learning features to detect them. Hongyu et al. [4] gathered spam messages into campaigns and used supervised machine learning framework for filtering them. Works in [1, 3, 21] studied spam by manually labeling their data sets into spam and non-spam accounts and built a classifier using account-based, content-based and behavior-based features. Classifiers of nature similar to these works can benefit from using features we found common across spam profiles in our data.

Significant research has also been done in detection and characterization of suspicious URLs in Twitter. Lee et al. [11] proposed a machine learning classifier to detect suspicious URLs by identifying characteristics of URLs in spam tweets. Thomas et al. [15] proposed a real time URL classifier that extracts features from page content, hosting infrastructure and lexical properties of URLs to detect spam urls in a Twitter stream.

Grier et al. [7] characterized Twitter spam by discussing various topics like type of tweets spammers use, click through rates of shortened URLs, types of spam accounts, blacklist performance and the techniques used by spammers to garner a wider audience.According to their analysis, 70% of spam tweets had hashtags (compared to 7.3% in our data), 11% were retweets (compared to 1.1% in our data) and 10.6% (compared to 5.1% in our data) were mentions.

Yang et al. [23] studied how spammers get embedded deeper in social networks and found three categories of users that form closer relationships with spammers that post malicious links. Jonghyuk et al. [12] proposed a spam filtering technique by analyzing sender-receiver relationship between users.

A very closely related work to our work here was done by Thomson et al. [16]. The authors collect a large number of suspended Twitter accounts over a period of time and analyze their behavior. Although we have a similar data collection methodology (albeit for a shorter period of time), we show how much Twitter spam has evolved since their study, likely to counter Twitter's current anti-spam efforts. Specifically, we find that multiple characteristics of spam accounts, such as social relationships formed, longevity and type of tweets used differ significantly in our study even though the nature of spam campaigns remains essentially the same. We focus on spam accounts which survive much longer than those set up by naive spammers and discuss their strategies.

8. CONCLUSION

We analyzed strategies of successful Twitter spammers in this paper, particularly as they relate to picking spam targets. A key finding of our work was that while spam campaigns on Twitter changed little, the spammers themselves evolved in a mere matter of one year since Thomas et al., perhaps to stay afloat amidst take-down efforts. The

complexity of their strategies are only likely to increase as more and more tools which simulate normal human behavior are developed. Given the shift in spammer behavior to integrate more closely into the social network, this will require constant scrutiny on the part of Twitter to discover newer and more complex strategies. We believe there is a need for spam classifiers to include social metadata such as follower metadata, keywords cloud, domains linked in tweets etc. along with the traditional signals used in classifiers today to achieve true success in identifying sophisticated spam profiles.

Acknowledgements

We thank Fil Menczer and the Center for Complex Networks and System Research (CNetS) at Indiana University for providing us access to the Twitter streaming API data through their Truthy project. The Truthy project and its infrastructure are supported by the National Science Foundation (NSF) grants CCF-1101743 and IIS-0811994 respectively.

The work in this paper is supported by the National Science Foundation (NSF) under Grant Number CNS-1018617. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

9. REFERENCES

[1] Benevenuto, F., Magno, G., Rodrigues, T., and Almeida, V. Detecting spammers on Twitter. In *Collaboration, Electronic messaging, Anti-Abuse and Spam Conference (CEAS)* (2010).

[2] Benevenuto, F., Rodrigues, T., Almeida, V., Almeida, J., and Chao Zhang, K. R. Identifying video spammers in online social networks. In *Workshop on Adversarial Information Retrieval on the Web (AirWeb), held in conjunction with the International World Wide Web (WWW) conference* (2008).

[3] Chu, Z., Gianvecchio, S., and Wang, H. Who is tweeting on Twitter: Human, bot, or cyborg? In *Annual Computer Security Applications Conference (ACSAC)* (2010).

[4] Gao, H., Chen, Y., Lee, K., Palsetia, D., and Choudhary, A. Towards online spam filtering in social networks. In *ISOC Network and Distributed System Security Symposium (NDSS)* (2012).

[5] Gao, H., Hu, J., Wilson, C., Li, Z., Chen, Y., and Zhao, B. Y. Detecting and characterizing social spam campaigns. In *ACM/USENIX Internet Measurement Conference (IMC)* (2010).

[6] Ghosh, S., Viswanath, B., Kooti, F., Sharma, N. K., Korlam, G., Benevenuto, F., Ganguly, N., and Gummadi, K. P. Understanding and combating link farming in the Twitter social network. In *International Conference on World Wide Web (WWW)* (2012).

[7] Grier, C., Thomas, K., Paxson, V., and Zhang, M. @spam: the underground on 140 characters or less. In *ACM Conference on Computer and Communications Security (CCS)* (2010).

[8] Jones, K. S. A statistical interpretation of term specificity and its application in retrieval. In *Journal of Documentation, Vol. 28 Issue: 1, pp.11 - 21* (1972).

[9] Lee, K., Caverlee, J., Kamath, K. Y., and Cheng, Z. Detecting collective attention spam. In *Workshop on WebQuality, held in conjunction with International World Wide Web (WWW) conference* (2012).

[10] Lee, K., Caverlee, J., and Webb, S. Uncovering social spammers: Social honeypots + machine learning. In *ACM Special Interest Group on Information Retrieval (SIGIR) Conference* (2010).

[11] Lee, S., and Kim, J. Warningbird: Detecting suspicious URLs in twitter stream. In *ISOC Network and Distributed System Security Symposium (NDSS)* (2012).

[12] Song, J., Lee, S., and Kim, J. Spam filtering in twitter using sender-receiver relationship. In *International Symposium on Recent Advances in Intrusion Detection (RAID)* (2011).

[13] Stringhini, G., Egele, M., Kruegel, C., and Vigna, G. Poultry markets: On the underground economy of Twitter followers. In *ACM Workshop on Online Social Networks (WOSN)* (2012).

[14] Stringhini, G., Kruegel, C., and Vigna, G. Detecting spammers on social networks. In *Annual Computer Security Applications (ACSAC) conference* (2010).

[15] Thomas, K., Grier, C., Ma, J., Paxson, V., and Song, D. Design and evaluation of a real-time URL spam filtering service. In *IEEE Symposium on Security and Privacy* (2011).

[16] Thomas, K., Grier, C., Song, D., and Paxson, V. Suspended accounts in retrospect: an analysis of twitter spam. In *ACM/USENIX Internet Measurement Conference (IMC)* (2011).

[17] TweetAdder, 2012. `http://www.tweetadder.com`.

[18] TweetAttacks manual, 2012. `http://www.scribd.com/doc/59395233/Manual-Tweet-Attacks`.

[19] Twitter rules, 2012. `https://support.twitter.com/entries/18311-the-twitter-rules`.

[20] Twitter size, 2012. `http://blog.twitter.com/2012/03/twitter-turns-six.html`.

[21] Wang, A. H. Don't follow me: Spam detection in Twitter. In *International Conference on Security and Cryptography (SECRYPT)* (2010).

[22] Yang, C., Harkreader, R., and Gu, G. Die free or live hard? Empirical evaluation and new design for fighting evolving Twitter spammers. In *International Symposium on Recent Advances in Intrusion Detection (RAID)* (2011).

[23] Yang, C., Harkreader, R., Zhang, J., Shin, S., and Gu, G. Analyzing spammers' social networks for fun and profit: A case study of cyber criminal ecosystem on Twitter. In *International Conference on World Wide Web (WWW)* (2012).

All Your Face Are Belong to Us: Breaking Facebook's Social Authentication

Iasonas Polakis*
FORTH-ICS, Greece
polakis@ics.forth.gr

Marco Lancini
Politecnico di Milano, Italia
marco.lancini@mail.polimi.it

Georgios Kontaxis
Columbia University, USA
kontaxis@cs.columbia.edu

Federico Maggi
Politecnico di Milano, Italia
fmaggi@elet.polimi.it

Sotiris Ioannidis
FORTH-ICS, Greece
sotiris@ics.forth.gr

Angelos D. Keromytis
Columbia University, USA
angelos@cs.columbia.edu

Stefano Zanero
Politecnico di Milano, Italia
zanero@elet.polimi.it

ABSTRACT

Two-factor authentication is widely used by high-value services to prevent adversaries from compromising accounts using stolen credentials. Facebook has recently released a two-factor authentication mechanism, referred to as Social Authentication, which requires users to identify some of their friends in randomly selected photos. A recent study has provided a formal analysis of social authentication weaknesses against attackers inside the victim's social circles. In this paper, we extend the threat model and study the attack surface of social authentication in practice, and show how any attacker can obtain the information needed to solve the challenges presented by Facebook. We implement a proof-of-concept system that utilizes widely available face recognition software and cloud services, and evaluate it using real public data collected from Facebook. Under the assumptions of Facebook's threat model, our results show that an attacker can obtain access to (sensitive) information for at least 42% of a user's friends that Facebook uses to generate social authentication challenges. By relying solely on publicly accessible information, a casual attacker can solve 22% of the social authentication tests in an automated fashion, and gain a significant advantage for an additional 56% of the tests, as opposed to just guessing. Additionally, we simulate the scenario of a determined attacker placing himself inside the victim's social circle by employing dummy accounts. In this case, the accuracy of our attack greatly increases and reaches 100% when 120 faces per friend are accessible by the attacker, even though it is very accurate with as little as 10 faces.

Categories and Subject Descriptors

K.6.5 [**Security and Protection**]: Authentication

General Terms

Security

Keywords

Authentication, Face recognition, Online social networks

*Iasonas Polakis and Sotiris Ioannidis are also with the University of Crete

ACSAC '12 Dec 3-7, 2012, Orlando, Florida USA

1. INTRODUCTION

Online social networks (OSNs) have become some of the fastest growing Web services with a massive user base and, at the same time, an appealing target for malicious activities: Facebook reports over 900 million active users as of March 2012, all the while encouraging its users to share more and more information online for a richer experience. Such accumulated data and the interconnections between users have made OSNs an attractive target for the Internet miscreants, which harvest account credentials using both technical and social-engineering attacks. Studies [22] have shown that traditional underground economies have shifted their focus from stolen credit card numbers to compromised social network profiles, which are sold for the highest prices. A recent study [12] reports that the vast majority of spamming accounts in OSNs are not dummy profiles created by attackers, but legitimate, existing user accounts that have been compromised. Additionally, new Facebook phishing attacks use compromised accounts to steal personal information [15].

As a standard method for strengthening the security of online user accounts, high-value services such as online banking, and recently Google services, have adopted two-factor authentication where users must present two separate pieces of evidence in order to authenticate. The two factors are such that the risk of an adversary acquiring both is very low. Typically, the two factors consist of something the user knows (e.g., a password) and something the user possesses (e.g., a hardware token). Physical tokens, however, are inconvenient for users, who may not always carry them, and costly for the service that deploys them.

In 2011 Facebook, in an effort to combat stolen account passwords, introduced its so-called Social Authentication (SA), a second authentication factor based on user-related social information that an adversary "half way around the world" supposedly lacks and cannot easily trick the owners into divulging. Following the standard password-based authentication, if Facebook deems it necessary, users are presented with photos of 7 of their friends and are asked to identify them. SA appears to be more user-friendly and practical as (i) users are required to identify photos of people

they know and (ii) they are accustomed to tagging photos of their friends—thus implicitly providing the necessary labeled dataset for Facebook.

In this paper we identify the vulnerable nature of SA and empirically confirm a series of weaknesses that enable an adversary to carry out an effective automated attack against Facebook's SA. The key of SA is the knowledge a user has about his online social circle, whereas an attacker trying to log into the account with stolen credentials, lacks. Facebook acknowledges that its heuristics and threat model do not cover the case of friends and family (i.e., anyone inside a user's online social circle) hacking into one's account. The intuition behind our research is that any stranger who obtains a user's password can gain enough data to defeat the SA mechanism.

To this end, we initially conduct a series of experiments to validate our assumptions about the access that an adversary might have to such information. The core of this paper is the design and implementation of an automated, modular system that defeats Facebook's SA mechanism. The general principles of our approach allow it to be extended and applied to any photo-based SA system. Initially, during a preparatory reconnaissance phase we obtain a victim's list of friends and the photos accessible from his OSN profile. This includes crawling the publicly-accessible portion of the victim's social graph and (optionally) performing actions that bring us inside the restricted part of the social circle, such as issuing friendship requests to the victim's friends. We then process the collected photos using face detection and recognition software to build each friend's facial model. An attacker is highly unlikely to be familiar with the friends of a victim—at least under the threat model assumed by Facebook—and there lies the security of recognizing one's friends as a security mechanism. However, by acquiring accurate facial models of a victim's friends we are in possession of the key to solving SA challenges. When the SA test is triggered, we lookup the identity of the depicted friends and provide an answer.

At a first glance, it might seem that our attack only affects Facebook users that leave their friends list and published photos publicly accessible. According to Dey R. et al. [9] (2012), 47% percent of Facebook users leave their friends list accessible by default. However, an attacker can always attempt to befriend his victims, thus gaining access to their protected information. Such actions may achieve up to a 90% success rate [4, 5, 19, 24]. That way, the set of vulnerable users may reach 84% of the Facebook population. At the same time, our experiments show that 71% of Facebook users expose at least one publicly-accessible photo album. Similarly, an attacker has very good chances of getting access, through online friendship requests, to profiles with private photo albums. Moreover, even if user A's photos are protected from public view and A does not accept friend requests from unknown people, user B might have a photo of A in which A is tagged (i.e., their face framed and labeled with his real name and Facebook ID). If user B has their photos public, A's tags are implicitly exposed to crawling. Overall, dynamics of OSNs such as Facebook, make it very hard for users to control their data [18, 23] and thereby increase the attack surface of threats against SA. We show that anyone can gain access to crucial information for at least 42% of the tagged friends used to build SA challenges that will protect a user's profile.

Under such minimal attack-surface assumptions we manually verify that our implemented SA breaker, powered by a face recognition module, solves 22% of the real SA tests presented by Facebook (28 out of 127 tests), in less than 60 seconds for each test. Moreover, our attack gives a significant advantage to an attacker as it solves 70% of each test (5 out of 7 pages) for 56% of the remainder tests (71 out of 99 tests). Note that we obtain this accuracy in real-world conditions by relying solely on publicly-available information, which anyone can access: We do not send friendship requests to the victims or their friends to gain access to more photos. Furthermore, our simulations demonstrate that within a maximized attack surface (i.e., if a victim, or one of his friends, accepts befriend requests from an attacker, which happens in up to 90% of the cases), the success rate of our attack increases to 100%, with as little as 120 faces per victim for training, and takes about 100 seconds per test.

A recent study [17], provided a formal analysis of the social authentication weaknesses against attacker within the victim's social circle. We expand the threat model and demonstrate in practice that any attacker, inside and outside the victim's social circle, can carry out automated attacks against the SA mechanism in an efficient manner. Therefore we argue that Facebook should reconsider its threat model and re-evaluate this security mechanism.

In summary, the key contributions of this work are the following:

- We systematize and expand previous work, which pointed out (i) the feasibility of recognizing people's faces using Facebook photos, and (ii) the theoretical issues with face-based SA. This systematization allows us to describe an attack that breaks Facebook's SA mechanism, while retaining the assumptions of their threat model.
- We present our black-box security analysis of Facebook's SA mechanism and point out its weaknesses (and implementation flaws) when employed as the second factor of a two-factor authentication scheme.
- We design and implement an automated, modular system that leverages face detection and recognition to break Facebook's SA efficiently. We, thus, show the feasibility of such an attack in large-scale.
- We show that publicly-available face recognition services offer a very accessible and precise alternative to building a custom face recognition system.
- We discuss how Facebook's SA scheme should be modified so that users can trust it as a second authentication factor.

2. SOCIAL AUTHENTICATION

We hereby describe the nature of Facebook's SA in terms of functionality and heuristics. We go beyond a general description and evaluate its behavior under real-world conditions. Facebook's SA was announced in January 2011 and, to the best of our knowledge, is the first instance of an authentication scheme based on the "who you know" rationale: A user's credentials are considered authentic only if the user can correctly identify his friends.

2.1 How Social Authentication Works

After the standard, password-based authentication, the user is presented with a sequence of 7 pages featuring authentication challenges. As shown in Fig. 1, each challenge is comprised of 3 photos of an online friend; the names of 6 people from the user's social circle are listed and he has to select the one depicted. The user is allowed to fail in 2 challenges, or skip them, but must correctly identify the people in at least 5 to pass the SA test.

2.2 Requirements for Triggering

Based on our analysis, Facebook activates the SA only for the fraction of accounts that have enough friends with a sufficient amount of tagged photos that contain a human face.

Friend list. SA requires that the user to be protected has a reasonable number of friends. From our experiments we have concluded that, in the case of Facebook, a user must have at least 50

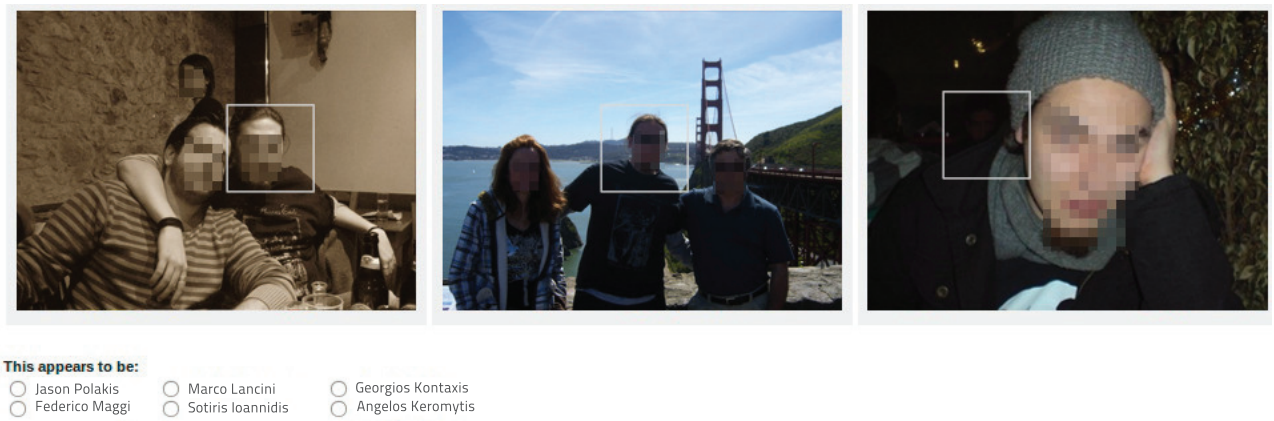


Figure 1: Example screenshot of the user interface of a Facebook SA page. The screenshot is synthetic due to copyright reasons, but is an exact replica of a real-world Facebook SA page. Faces have been pixelated for privacy reasons.

friends. To obtain this information, we created 11 distinct dummy profiles and increased the number of friends of these accounts on a daily basis, until we managed to trigger the SA (detailed in §4.3).

Tagged photos. The user's friend must be tagged (placed in a labeled frame) in an adequate number of photos. Keep in mind that since these are user-submitted tags, Facebook's dataset can get easily tainted. People often erroneously tag funny objects as their friends or publish photos with many friends tagged, several of whom may not actually be present in the photo.

Faces. SA tests must be solvable by humans within the 5 minute (circa) time window enforced by Facebook. We argue that Facebook employs a face detection algorithm to filter the dataset of tagged people to select photos with tagged faces. From our manual inspection of 127 instances of real SA tests (2,667 photos), we have noticed that Facebook's selection process is quite precise, despite some inaccuracies that lead to SA tests where some photos contain no face. Overall, 84% of these 2,667 photos contained at least one human-recognizable face, and about 80% of them contained at least one face such that an advanced face detection software can discern—in this test, we used `face.com`. To validate our argument on the use of face detection filtering, we repeated the same manual inspection on a different set of 3,486 photos drawn at random from our dataset of 16,141,426 photos (detailed in §4.1). We then cropped these images around the tags; hence, we generated a SA dataset in the same manner that Facebook would if it naively relied only on people's tagging activity. Only 69% ($< 84\%$) of these photos contain at least one recognizable human face, thus the baseline number of faces per tag is lower in general than in the photos found in the real SA tests. This confirms our hypothesis that Facebook employs filtering procedures to make sure each SA test page shows the face of the person in question in at least one photo.

Triggering. Facebook triggers the SA when it detects a suspicious login attempt, according to a set of heuristics. Our experiments reveal that this happens when (i) the user logs in from a different geographical location, or (ii) uses a new device (e.g., computer or smartphone) for the first time to access his account.

2.3 Advantages and Shortcomings

The major difference from the traditional two-factor authentication mechanisms (e.g., confirmation codes sent via text message or OTP tokens) is that Facebook's SA is less cumbersome, especially because users have grown accustomed to tagging friends in photos. However, as presented recently by Kim et al. [17], designing a usable yet secure SA scheme is difficult in tightly-connected social graphs, not necessarily small in size, such as university networks.

Our evaluation suggests that SA carries additional implementation drawbacks. First of all, the number of friends can influence the applicability and the usability of SA. In particular, users with many friends may find it difficult to identify them, especially when there are loose or no actual relationships with such friends. A typical case is a celebrity or a public figure. Even normal users, with 190 friends on average[1], might be unable to identify photos of online contacts that they do not interact with regularly. Dunbar's number [11] suggests that humans can maintain a stable social relationship with at most 150 people. This limit indicates a potential obstacle in the usability of the current SA implementation, and should be taken into account in future designs.

Another parameter that influences the usability of SA is the number of photos that depict the actual user, or at least that contain objects that uniquely identify the particular user. As a matter of fact, feedback [15] from users clearly expresses their frustration when challenged by Facebook to identify inanimate objects that they or their friends have erroneously tagged for fun or as part of a contest which required them to do so.

Finally, in certain cases, Facebook currently presents users with the option to bypass the SA test by providing their date of birth. This constitutes a major flaw in their security mechanism. Obtaining the victim's date of birth is trivial for an adversary, as users may reveal this information on their Facebook profile.

2.4 Threat Model and Known Attacks

Throughout this paper we refer to the people inside a user's online social circle as friends. Friends have access to information used by the SA mechanism. Tightly-connected social circles where a user's friends are also friends with each other are the worst scenarios for SA, as potentially any member has enough information to solve the SA for any other user in the circle. However, Facebook designed SA as a protection mechanism against strangers, who have access to none or very little information. Under this threat model, strangers are unlikely to be able to solve an SA test. We argue that any stranger can position himself inside the victim's social circle, thereby gaining the information necessary to defeat the SA mechanism. Kim et al. [17] suggest that the progress made by face-recognition techniques may enable automated attacks against photo-based authentication mechanisms. At the same time, Dantone et al. [8] have demonstrated that social relationships can also be used to improve the accuracy of face recognition. Moreover, Ac-

[1] `https://www.facebook.com/notes/facebook-data-team/anatomy-of-facebook/10150388519243859`

quisti et al. [1] went beyond the previous approach and presented a system that can associate names to faces and, thus, de-anonymize a person solely by using a picture of his or her face. Although no scientific experimentation on real-world data has been made to measure the weakness of SA, these studies suggest that the face-to-name relation, which is the security key behind SA, may be exploited further to demonstrate that the scheme is insecure. Our intuition that attackers can overcome the limitations of Facebook's perceived threat model has been the motivation behind this paper.

2.5 Attack Surface Estimation

In our attack model, the attacker has compromised the user's credentials. This is not an unreasonable assumption; it is actually the reason behind the deployment of the SA. This can be accomplished in many ways (e.g., phishing, trojan horses, key logging, social engineering) depending on the adversary's skills and determination [10]. Statistically speaking, our initial investigation reveals that Facebook's current implementation results in 2 out of 3 photos of each SA page (84% of 3 is 2.523) with at least one face that a human can recognize. This makes SA tests solvable by humans. However, our investigation also reveals that about 80% of the photos found in SA tests contain at least one face that can be detected by face-detection software. This rationale makes us argue that an automated system can successfully pass the SA mechanism. To better understand the impact of our attack, we provide an empirical calculation of the probabilities of each phase of our attack. In other words, if an attacker has obtained the credentials of any Facebook user, what is the probability that he will be able to access the account? What is the probability if he also employs friend requests to access non-public information on profiles? To derive the portion of users susceptible to this threat, we built the attack tree of Fig. 2.

We distinguish between a casual and a determined attacker, where the former leverages publicly-accessible information from a victim's social graph whereas the latter actively attempts to gather additional private information through friendship requests.

Friends list. Initially, any attacker requires access to the victim's friends list. According to Dey et al. [9] $\mathrm{P}(F) = 47\%$ of the user's have their friends list public—as of March 2012. If that is not the case, a determined attacker can try to befriend his victim. Studies have shown [4, 5, 19, 24] that a very large fraction of users tends to accept friend requests and have reported percentages with a 60–90% chance of succeeding (in our analysis we use 70%, lower than what the most recent studies report). Therefore, he has a combined 84% chance of success so far, versus 47% for the casual attacker.

Photos. Ideally the attacker gains access to all the photos of all the friends of a victim. Then with a probability of 1 he can solve any SA test. In reality, he is able to access only a subset of the photos from all or a subset of the friends of a victim. Our study of 236,752 Facebook users revealed that $P(P) = 71\%$ of them exposed at least one public photo album. Again we assume that a determined attacker can try to befriend the friends of his victim to gain access to their private photos with a chance of $P(B) \simeq 70\%$ to succeed, which is a conservative average compared to previous studies. At the end of this step, the determined attacker has on average at least one photo for 77% of the friends of his victim while a casual attacker has that for 33%. This is versus Facebook which has that for 100% of the friends with uploaded photos.

Tags. The next step is to extract labeled frames (tags) of people's faces from the above set of photos to compile $\langle \texttt{uid}, \texttt{face} \rangle$ tuples used by Facebook to generate SA tests and by the attacker to train facial models so as to respond to those tests. By analyzing 16,141,426 photos from out dataset, corresponding to the 33% of friends' photos for the casual attacker, we found that 17% of these photos contain tags (hence usable for generating SA tests), yet only the 3% contain tags about the owner of the photo. This means that by crawling a profile and accessing its photos it is more likely to get tags of friends of that profile than of that profile itself. The astute reader notices that Facebook also has to focus on that 17% of photos containing tags to generate SA tests: Facebook will utilize the 17% containing tags of all the photos uploaded by a user's friends and therefore generate SA tests based on 100% of the friends for whom tags are available, whereas an attacker usually has access to less than that. In the extreme case, having access to a single friend who has tagged photos of all the other friends of the target user (e.g., he is the "photographer" of the group), the attacker will acquire at least one tag of each friend of the user and will be able to train a face recognition system for 100% of the subjects that might appear in an SA test. In practice, by collecting the tags from the photos in our dataset we were able to gather $\langle \texttt{uid}, \texttt{face} \rangle$ tuples for 42% of the people in the friend lists of the respective users. Therefore, assuming that all of a user's friends have tagged photos of them on Facebook, a casual attacker is able to acquire this sensitive information for 42% of the tagged friends used by Facebook to generate SA tests. As we show in §4.3, with only that amount of data, we manage to automatically solve 22% of the real SA tests presented to us by Facebook, and gain a significant advantage for an additional 56% with answers to more than half the parts of each test. We cannot calculate the corresponding percentage for the determined attacker without crawling private photos (we discuss the ethical reasons for this in §5). However, we simulate this scenario in §4.2 and find that we are able to pass the SA tests on average with as little as 10 faces per friend.

Faces. Finally, from the tagged photos, the attacker has to keep the photos that actually feature a human face and discard the rest—we can safely hypothesize Facebook does the same, as discussed in §2.2. We found that 80% of the tagged photos in our dataset contain human faces that can be detected by face-detection software, and Facebook seems to follow the same practice; therefore, the advantage for either side is equal. Overall, our initial investigation reveals that up to 84% of Facebook users are exposed to the crawling of their friends and their photos. They are, thus, exposed to attacks against the information used to protect them through the SA mechanism. A casual attacker can access $\langle \texttt{uid}, \texttt{face} \rangle$ tuples of at least 42% of the tagged friends used to generate social authentication tests for a given user. Such information is considered sensitive, known only to the user and the user's circle, and its secrecy provides the strength to this mechanism.

3. BREAKING SOCIAL AUTHENTICATION

Our approach applies to any photo-based SA mechanism and can be extended to cover other types of SA that rely on the proof of knowledge of "raw" information (e.g., biographies, activities, relationships and other information from the profiles of one's social circle). We focus on Facebook's SA, as it is the only widespread and publicly-available deployment of this type of social authentication. As detailed in §3.1, our attack consists of three preparation steps (steps 1-3), which the attacker runs offline, and one execution step (step 4), which the attacker runs in real-time when presented with the SA test. Fig. 3 presents an overview of our system's design.

3.1 Implementation Details

3.1.1 Step 1: Crawling Friend List

Given the victim's UID, a crawler module retrieves the UIDs and names of the victim's friends and inserts them in our database. As discussed in §2.5, casual attackers can access the friend list when this is publicly available (47% of the users), whereas determined attackers can reach about 84% of the friend lists by issuing befriend

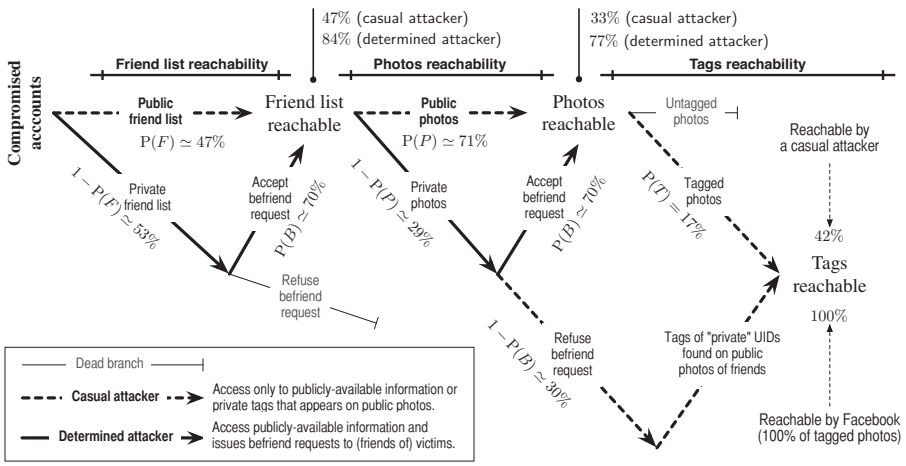


Figure 2: Attack tree to estimate the vulnerable Facebook population. Not all the branches are complete, as we consider only the events that are relevant to the case study.

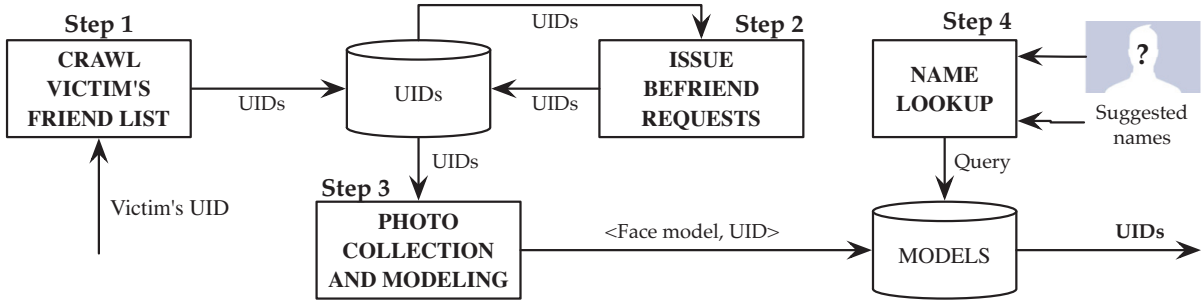


Figure 3: Overview of our automated SA-breaking system. It operates in four steps. In Step 1 we retrieve the victim's friend list using his or her UID. Then, in Step 2 (optional), we send befriend requests, so that we have more photos to extract faces from and build face classifiers in Step 3. In Step 4, given a photo, we query the models to retrieve the corresponding UID and thus match a name to face.

requests. We implement the crawling procedures using Python's `urllib` HTTP library and regular expression matching to scrape Facebook pages and extract content. We store the retrieved data in a MongoDB database, a lightweight, distributed document-oriented storage suitable for large data collections, and keep the downloaded photos in its GridFS filesystem.

3.1.2 Step 2: Issuing Friend Requests

An attacker can use legitimate-looking, dummy profiles to send friendship requests to all of the victim's friends. As shown in Fig. 2, this step can expand the attack surface by increasing the reachable photos. We implement a procedure that issues befriend requests via the fake accounts we have created for our experimental evaluation (see §4.1). Even though we do not collect any private information or photos of these users for our experiments, we need an adequate number of friends in our accounts to be able to trigger the SA mechanism. We select users for our requests, based on the friends suggested by Facebook. Also, as shown by Irani et al. [14], to achieve a high ratio of accepted friend requests, we create profiles of attractive women and men with legitimate-looking photos[2] (i.e., avoiding the use of provocative or nudity photos). In addition, we inject some random profile activity (e.g., status messages, like activities). If Facebook triggers CAPTCHA challenges at some point, our system prompts a human operator to intervene. However, Bilge et al. [4] have demonstrated the use of automated systems against the CAPTCHA countermeasure. Moreover, to hinder spammers, Facebook limits the number of friend requests each profile is allowed to issue in a short period of time and enforces a "cooldown" period of two days on misbehavior. To overcome this obstacle and still have profiles with an adequate amount of friends, we spread our friend requests over a period of one week. We also noticed that for profiles that have education and employment information and send requests to people within these circles, Facebook enforces more relaxed thresholds and allowed us to send close to 100 requests in a single day. In addition, the method described by Irani et al. [14] allows to increase the number of friends passively as opposed to requesting friendships explicitly.

3.1.3 Step 3: Photo Collection/Modeling

Step 3.1: Photo collection We collect the URLs of all the photos contained in the albums of the target's friends using the same screen-scraping approach that we described in Step 3.1.1. We then feed the collected URLs into a simple module that does the actual download. This module stores in the database the metadata associated with each downloaded photo: URL, UID of the owner, tags and their coordinates (in pixels).

[2]We selected photos from a database of models.

Step 3.2: Face Extraction and Tag Matching We scan each downloaded photo to find faces. Specifically, we use a face detection classifier part of the OpenCV toolkit[3]. There are plenty of face detection techniques available in the literature, more precise than the one that we decided to use. However, our goal is to show that face-based SA offers only a weak protection, because even with simple, off-the-shelf tools, an adversary can implement an automated attack that bypasses it.

Subsequently, we label each face with the UID of the nearest tag found in the adjacent 5%-radius area, calculated with the euclidean distance between the face's center and the tag's center. Unlabeled faces and tags with no face are useless, thus we discard them. We save the selected faces as grayscale images, one per face, resized to 130×130 pixels.

Step 3.3: Facial Modeling We use the `sklearn` library[4] to construct a supervised classifier. We first preprocess each face via histogram equalization to ensure uniform contrast across all the samples. To make the classification job feasible with these many features (i.e., 130×130 matrices of integers), we project each matrix on the space defined by the 150 principal components (i.e., the "eigenfaces"). We tested K-nearest-neighbors (kNN), tree, and support-vector (with a radial-basis kernel) classifiers using a K-fold cross-validation technique. We found that support-vector classifiers (SVC) yield the highest accuracy, but are very expensive computationally. Therefore, we use kNN classifiers, with $k = 3$ as they provide a faster alternative to SVC with comparable accuracy.

3.1.4 Step 4: Name Lookup

When Facebook challenges our system with a SA test, we submit the photos from the SA test to the classifier, which attempts to identify the depicted person and select the correct name. We detect the faces in each of the 7 photos of an SA page and extract the 150 principal components from each face's 130×130 matrix. Then, we use the classifier to predict the class (i.e., the UID) corresponding to each unknown face, if any. If, as in the case of Facebook, a list of suggested names (i.e., UIDs) is available, we narrow its scope to these names. Then, we query the classifier and select the outcome as the correct UID for each unknown face, choosing the UID that exhibits more consensus (i.e., more classifiers output that UID) or the highest average prediction confidence.

3.2 Face Recognition as a Service

Automatic face recognition is approaching the point of being ubiquitous: Web sites require it and users expect it. Therefore, we investigate whether we can employ advanced face recognition software offered as a cloud service. We select `face.com` which offers a face recognition platform for developers to build their applications on top of. Incidentally, `face.com` was recently acquired by Facebook[5]. The service exposes an API through which developers can supply a set of photos to use as training data and then query the service with a new unknown photo for the recognition of known individuals. The training data remains in the cloud. Developers can use up to two different namespaces (i.e., separate sets of training data) each one able to hold up to 1,000 users, where each user may be trained with a seemingly unbound number of photos. Usage of the API is limited to 5,000 requests an hour. Such a usage framework may be restrictive for building popular applications with thousands of users but it is more than fitting for the tasks of an adversary seeking to defeat photo-based social authentication. Assuming the free registration to the service, one may create a training set for up to 1,000 of a victim's friends (the max limit for Facebook is 5,000 although the average user has 190 friends). After that, one can register more free accounts or simply delete the training set when no longer necessary and reclaim the namespace for a new one. We develop a proof-of-concept module for our system that leverages the `face.com` API as an alternative, service-based implementation of steps 3 and 4 from Fig. 3. We submit the photos to the service via the `faces.detect` API call to identify any existing faces and determine whether they are good candidates for training the classifier. The next step is to label the good photos with the respective UIDs of their owners (`tags.save`). Finally we initiate the training on the provided data (`faces.train`) and once the process is complete we can begin our face recognition queries—the equivalent of step 4 from Fig. 3. Once the training phase is finished, the service is able to respond within a couple of seconds with a positive or negative face recognition decision through the `faces.recognize` call. We take advantage of the ability to limit the face matching to a group of uids from the training set and we do so for the suggested names provided by Facebook for each SA page.

[3] `http://opencv.itseez.com/`

[4] `http://scikit-learn.org`

[5] `http://face.com/blog/facebook-acquires-face-com/`

	TOTAL	PUBLIC	PRIVATE
UIDs	236,752	167,359	69,393
Not tagged	116,164	73,003	43,161
Tagged	120,588	94,356	26,232
Mean tags per UID:		19.39	10.58
Tags[9]	2,107,032	1,829,485	277,547
Photos	16,141,426	16,141,426	(not collected)
Albums	805,930	805,930	(not collected)

Table 1: Summary of our collected dataset. The terms "public", and "private" are defined in §4.1.

4. EXPERIMENTAL EVALUATION

Here we evaluate the nature of Facebook's SA mechanism and our efforts to build an automated SA solving system. We first assess the quality of our dataset of Facebook users (§4.1). We consider this a representative sample of the population of the online social network. We have not attempted to compromise or otherwise damage the users or their accounts. We collected our dataset as a casual attacker would do. Next we evaluate the accuracy and efficiency of our attack. In §4.2 we use simulation to play the role of a determined attacker, who has access to the majority of the victims' photos. In §4.3 we relax this assumption and test our attack as a casual attacker, who may lack some information (e.g., the victims may expose no photos to the public, there are no usable photos, no friend requests issued). More details on the capabilities of these two types of attacker are given in §2.5.

For part of our experiments we implemented custom face recognition software. This was done for two reasons. First, because we needed something very flexible to use, that allowed us to perform as many offline experiments as needed for the experiments of the determined attacker. Second, we wanted to show that even off-the-shelf algorithms were enough to break the SA test, at least in ideal conditions. However, superior recognition algorithms exist, and we conducted exploratory experiments that showed that face.com, although less flexible than our custom solution, has much better accuracy. Therefore, we decided to use it in the most challenging conditions, that is to break SA tests under the hypothesis of the casual attacker.

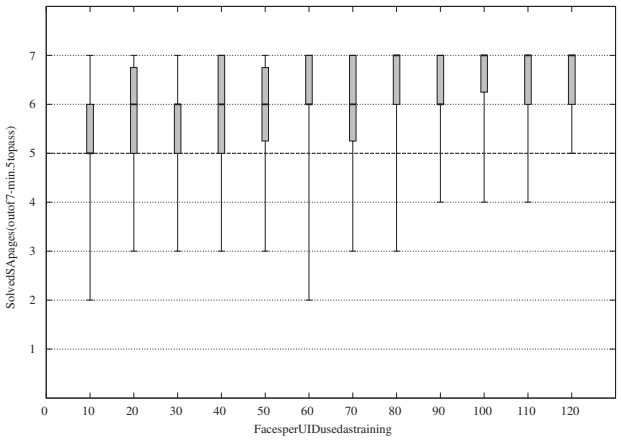


Figure 4: Percentage of successfully-passed tests as a function of the size of the training set. For each iteration, 30 randomly-generated offline SA tests were used.

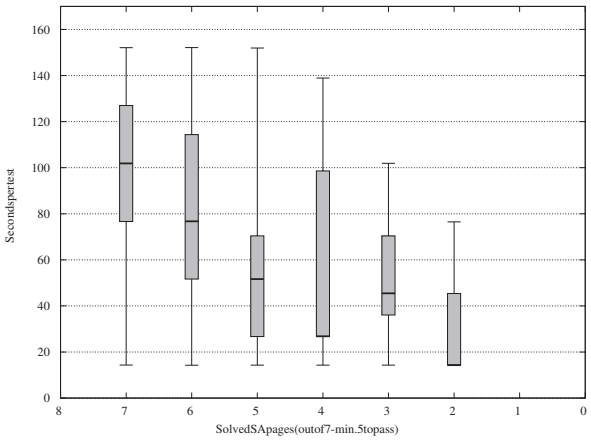


Figure 5: Time required to lookup photos from SA tests in the face recognition system.

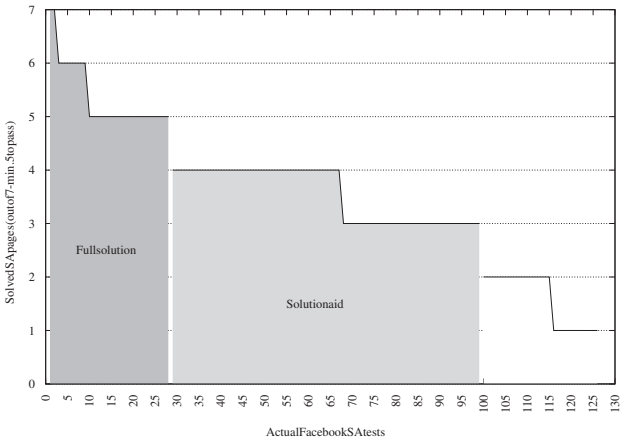


Figure 6: Efficiency of automated SA breaker against actual Facebook tests.

4.1 Overall Dataset

Our dataset contains data about real Facebook users, including their UIDs, photos, tags, and friendship relationships, as summarized in Table 1. Through public crawling we collected data regarding 236,752 distinct Facebook users. 71% (167,359) of them have at least one publicly-accessible album. We refer to these users as public UIDs (or public users). The remaining 29% of UIDS (69,393) keep their albums private (i.e., private UIDs, or private users). We found that 38% of them (26,232 or 11% of the total users) are still reachable because their friends have tagged them in one of the photos in their own profile (to which we have access). We refer to these UIDs as semi-public UIDs (or semi-public users). Data about the remaining 62% of UIDs (43,161 or 18% of the total users) is not obtainable because these users keep their albums private, and their faces are not found in any of the public photos of their friends. The public UIDs lead us to 805,930 public albums, totaling 16,141,426 photos and 2,107,032 tags that point to 1,877,726 distinct UIDs. It is therefore evident that people exposing (or making otherwise available) their photos are not only revealing information about themselves but also about their friends. This presents a subtle threat against these friends who cannot control the leakage of their names and faces. Albeit this dataset only covers a very small portion of the immense Facebook user base, we consider it adequate enough to carry out thorough evaluation experiments.

4.2 Breaking SA: Determined Attacker

The following experiment provides insight concerning the number of faces per user needed to train a classifier to successfully solve the SA tests. We create simulated SA tests using the following methodology. We train our system using a training set of $K = 10, 20, \ldots, 120$ faces per UID. We extract the faces automatically, without manual intervention, using face detection as described in §3.1.3. We then generate 30 SA tests. Each test contains 3 target photos per 7 pages showing the face of the same victim. The photos are selected randomly from the pool of public photos we have for each person, from which we exclude the ones used for the training. For each page and K we record the output of the name-lookup step (step 4), that is the prediction of the classifier as described in §3.1.4, and the CPU-time required. Fig. 4 shows the number of pages solved correctly out of 7, and Fig. 5 shows the CPU-time required to solve the full test (7 pages).

For an SA test to be solved successfully, Facebook requires that 5 out of 7 challenges are solved correctly. Our results show that our attack is always successful (i.e., at least 5 pages solved over 7) on average, even when a scarce number of faces is available. Clearly, having an ample training dataset such as $K > 100$ ensures a more robust outcome (i.e., 7 pages solved over 7). Thus, our attack is very accurate. As summarized in Fig. 5, our attack is also efficient because the time required for both "on the fly" training—on the K faces of the 6 suggested users—and testing remains within the 5-minute timeout imposed by Facebook to solve a SA test. An attacker may choose to implement the training phase offline using faces of all the victim's friends. This choice would be mandatory if Facebook—or any other Web site employing SA—decided to increase the number of suggested names, or remove them completely, such that "on the fly" training becomes too expensive.

4.3 Breaking SA: Casual Attacker

In the following experiment we assume the role of a casual attacker, with significantly more limited access to tag data for the training of a face recognition system. At the same time we attempt to solve real Facebook SA tests using the following methodology.

[9]On 11 April 2012, our crawler had collected 2,107,032 of such tags, although the crawler's queue contains 7,714,548 distinct tags.

We have created 11 dummy accounts that play the role of victims and populate them with actual Facebook users as friends and activity. Then, we employ a graphical Web browser scripted via Selenium[6] to log into these accounts in an automated fashion. To trigger the SA mechanism we employ Tor[7] which allows us to take advantage of the geographic dispersion of its exit nodes, thus appearing to be logging in from remote location in a very short time. By periodically selecting a different exit node, as well as modifying our user-agent identifier, we can arbitrarily trigger the SA mechanism. Once we are presented with an SA test, we iterate its pages and download the presented photos and suggested names, essentially taking a snapshot of the test for our experiments. We are then able to take the same test offline as many times necessary. Note that this is done for evaluation purposes and that the same system in production would take the test once and online. Overall, we collected 127 distinct SA tests.

We tried breaking the real SA tests using our module for `face.com` described in §3.2. Fig. 6 presents the outcome of the tests. Overall we are able to solve 22% of the tests (28/127) with people recognized in 5–7 of the 7 test pages and significantly improve the power of an attacker for 56% of the tests (71/127) where people were recognized in 3–4 of the 7 test pages. At the same time, it took 44 seconds on average with a standard deviation of 4 seconds to process the photos for a complete test (21 photos). Note that the time allowed by Facebook is 300 seconds.

We further analyzed the photos from the pages of the SA tests that failed to produce any recognized individual. In about 25% of the photos `face.com` was unable to detect a human face. We manually inspected these photos and confirmed that either a human was shown without his face being clearly visible or no human was present at all. We argue that humans will also have a hard time recognizing these individuals unless they are very close to them so that they can identify them by their clothes, posture or the event. Moreover, in 50% of the photos `face.com` was able to detect a human face but marked it as unrecognizable. This indicates that it is either a poor quality photo (e.g., low light conditions, blurred) or the subject is wearing sunglasses or is turned away from the camera. Finally, in the last 25% of the photos a face was detected but did not match any of the faces in our training set.

Overall, the accuracy of our automated SA breaker significantly aids an attacker in possession of a victim's password. A total stranger, the threat assumed by Facebook, would have to guess the correct individual for at least 5 of the 7 pages with 6 options per page to choose from. Therefore, the probability [8] of successfully solving an SA test with no other information is $O(10^{-4})$, assuming photos of the same user do not appear in different pages during the test. At the same time, we have managed to solve SA tests without guessing, using our system, in more than 22% of the tests and reduce the need to guess to only 1–2 (of the 5) pages for 56% of the tests, thus having a probability of $O(10^{-1})$ to $O(10^{-2})$ to solve those SA tests correctly. Overall in 78% of the real social authentication tests presented by Facebook we managed to either defeat the tests or offer a significant advantage in solving them.

After these experiments, we deleted all the photos collected from the real SA tests, as they could potentially belong to private albums of our accounts' friends, not publicly accessible otherwise.

[6] `http://seleniumhq.org`

[7] `http://www.torproject.org`

[8] Calculated using the binomial probability formula used to find probabilities for a series of Bernoulli trials.

5. ETHICAL CONSIDERATIONS

In this paper we explore the feasibility of automated attacks against the SA mechanism deployed by Facebook. As our experiments involve actual users, the question of whether this is ethically justifiable arises. We believe that research that involves the systematic exploration of real attacks is crucial, as it can reveal weaknesses and vulnerabilities in deployed systems, and provide valuable insight that can lead better solutions. This opinion is also shared among other security researchers [5, 16].

Nonetheless, we designed our experiments such that we minimize the impact of our research and preserve the privacy of these users. First, we never retained verbatim copies of sensitive information, besides the photos that we clearly needed for running the experiments. Secondly, our attack can optionally issue friend requests with the purpose of expanding the number of accessible photos. However, we issued friendship requests exclusively to reach the 50-friends threshold, required by Facebook to trigger the SA mechanism. We never took advantage of accepted requests to collect photos or other private information otherwise unavailable; we solely collected public photos. In particular, in §4.2 we simulated a determined attacker, by assuming he has obtained access to all the photos (both public and private) needed to launch the attacker under ideal conditions. We simulated these conditions using publicly-available photos.

6. REMEDIATION AND LIMITATIONS

Facebook has already devised some mechanisms that aim at hindering casual attackers and the practices presented in this paper. We explain why these mechanisms are not very effective or have some drawbacks that make them impractical. We continue with some proposed modifications to SA to make it safer based on the insights we obtained through our experiments.

6.1 Compromise Prevention and Notification

Facebook has recently deployed some security features that can help further defend against stolen credentials being used for compromising accounts. However, these mechanisms are opt-in and disabled by default. Therefore, users may not have them enabled, and will remain susceptible to the threat that we study in this paper.

First, users can add certain devices to a list of recognized, trusted devices. Whenever a user logs in from an unrecognized device, a security token is sent to the owner's mobile phone. This token must be entered in the log-in form for the user to be successfully logged in. This security setting, called login approval, follows the traditional second-token authentication scheme and only works in combination with the recognized, trusted devices feature. This approach can completely deter our attack, because it implements a truly-strong, two-factor authentication: The adversary would need physical access to the user's mobile phone to obtain the security token and successfully login.

Second, a user who fails to complete an SA challenge is redirected to an alert page, upon the next successful login, which reports the attempted login, and shows the time and place information. Unfortunately, if the adversary manages to solve the SA test in a subsequent attempt, he will be redirected to the notification page and the account owner will never see the alert. In addition to the default notification, users may enable an optional login-notification feature: Whenever their account is accessed, an alert message is sent via text or email message. This notification feature does not prevent an adversary from logging in and, therefore, does not prevent our attack, which takes less than one minute. Furthermore, if the adversary has compromised the email account—which is not an unrealistic assumption, as users may reuse their credentials across

services—he can delete the notification email. If that is not the case, the adversary will still have access until the owner changes the password and terminates any illegal active sessions.

Moreover, these mechanisms present three additional drawbacks. First, users must link their mobile phone number to their Facebook account, which many may not wish to do. Second, and more importantly, users typically access their account from many devices some of which may be public (e.g., computers in libraries or their workplace). In this case, adding all these devices to the list of trusted devices is both impractical and insecure, and users will not wish to receive alerts every time they log in from one of those machines. Finally, involving the cellular network may result in monetary charges, a factor which could seriously discourage users from opting in to the mechanism.

6.2 Slowing Down the Attacker

When the attacker is prompted with an SA challenge, he must solve a CAPTCHA before the actual SA test. Although this topic falls outside the scope of this paper, it is worth noticing that solving a CAPTCHA is trivial and only takes a human a few seconds. In addition, as previous work [4, 5, 7] has shown, breaking CAPTCHAs automatically is feasible and, in many cases, easy. Furthermore, it is well known that adversaries can perform laundry attacks [2, 13] and crowd-source the solution of CAPTCHAs. In conclusion, CAPTCHAs may create a technical obstacle to automated attacks, but they should not be considered a definitive countermeasure.

The presence of suggested names in SA tests is the major disadvantage of the current implementation as it greatly limits the search space for adversaries. By removing suggestions, there is a high probability of face-recognition software returning multiple users with similar confidence scores. Also, the time needed for face recognition might increase for certain systems although, as we have shown, cloud-based face recognition systems are unlikely to be seriously affected. On the downside, it will be harder for users to identify their friends and the system will be less usable as one would have to manually type the exact names of his friends. Automatic "type ahead" features may lessen the burden, although they are still vulnerable to exhaustive enumeration.

6.3 SA revisited

Designing effective and usable CAPTCHAs [6] is as hard as designing effective and usable authentication schemes that exploit social knowledge [17]. The downside of CAPTCHAs is that they are either too easy for machines or too difficult for humans. This paper and previous work show that the main weakness of social-based authentication schemes is that the knowledge needed to solve them is too public: Ironically, the purpose of social networks and the nature of human beings is to share knowledge. However, we believe that SA tests could be more secure yet still solvable by humans.

Facebook can build SA tests from photos showing human faces that fail or achieve very low confidence scores in Facebook's own face recognition system. Photos may contain faces of people wearing glasses, with masks on or slightly turned away from the camera. Humans are able to recognize their friends in the general image of a person, the environment, or the event. On the other hand, face-recognition algorithms have a hard time matching these human and social characteristics across different photos. In terms of feasibility, Facebook can piggy-back on users uploading photos as part of their daily routine and prompt them to tag a person for which no face has been detected, thus creating the necessary labeled dataset for generating SA tests. Even if an adversary is able to capture that photo and the tag provided by the user, chances are his face recognition algorithm will fail to find a resemblance with other photos of the same person. Also, if the adversary carries out an informed training process this might introduce unwanted noise which will reduce the overall accuracy of the classifier.

7. RELATED WORK

Previous work showed that information available in users' profiles in social networks can be used to break authentication mechanisms, or deduce information that threatens their privacy. A study performed by Rabkin [21] attempted to assess the security properties of personal knowledge questions that are used for fallback authentication. In §2.5 we discuss a similar study, although focused on Facebook SA. Rabkin argues that since such mechanisms owe their strength to the hardness of an information-retrieval problem, in the era of online social networks and the vast availability of personal information, their security is diminishing. In this study 12% of the sampled security questions from online banking sites is automatically attackable (i.e., the answers are on a user's profile).

Polakis et al. [20] demonstrate and evaluate how names extracted from OSNs can be used to harvest e-mail addresses as a first step for personalized phishing campaigns. Using information from Facebook, Twitter and Google Buzz, over 40% of the users' e-mail addresses could be directly mapped to their social profiles. A similar study was conducted by Balduzzi et al. [3]. They focus on the ability to use search utilities in social networking sites as an oracle; an attacker can search for an e-mail address, and if a user profile has been registered with that address, the profile is returned. Thus, the attacker can map e-mail addresses to social profiles.

The work most related to this paper is a recent study by Kim et al. [17], already discussed in §2.4. They formally quantify the advantage an attacker has against SA tests when he is already inside the victim's social circle. The researchers thus demonstrate that SA is ineffective against one's close friends and family or highly connected social sub-networks such as universities. However, in this paper we extend the threat model to incorporate any attacker located outside the victim's social circle. Furthermore, we implement a proof-of-concept infrastructure, and use publicly available information to quantify the effectiveness of such attacks. Thus, we are able to show the true extent to which SA is susceptible to automated attacks. Previous work [4, 5, 19, 24] has proved the feasibility of positioning one's self among a target's social circle using a mix of active and passive [14] techniques ranging from social engineering (e.g., attractive fake profiles) to forgetful users accepting friendship requests from fake profiles of people they are already linked. As such, the proposed countermeasures by Kim et al. [17] for a more secure social authentication mechanism remain equally vulnerable to our attack. Finally, we also present a theoretical estimation of the attack surface based on empirical data from our experiments as well as those reported by previous studies.

Boshmaf et al. [5] explore the feasibility of socialbots infiltrating social networks, and operate a Socialbot Network in Facebook for 8 weeks. A core aspect of their operation is the creation of new accounts that follow a stealthy behavior and try to imitate human users. These actions are complimentary to our attack, as a determined attacker can use them to infiltrate the social circles of his victims and expand the attack surface by gaining access to private photos. This will result in much higher percentages of solved SA tests. Gao et al. [12] found that 97% of malicious accounts were compromised accounts of legitimate users. This reflects the importance of social authentication as a mechanism for preventing attackers from taking over user accounts using stolen credentials. Accordingly, in this paper we explore the feasibility of an automated attack that breaks SA tests though the use of face recognition techniques. Our results validate the effectiveness of our attack even when the attacker uses only publicly available information.

8. CONCLUSIONS

In this paper we pointed out the security weaknesses of using social authentication as part of a two-factor authentication scheme, focusing on Facebook's deployment. We found that if an attacker manages to acquire the first factor (password), he can access, on average, 42% of the data used to generate the second factor, thus, gaining the ability to identify randomly selected photos of the victim's friends. Given that information, we managed to solve 22% of the real Facebook SA tests presented to us during our experiments and gain a significant advantage to an additional 56% of the tests with answers for more than half of pages of each test. We have designed an automated social authentication breaking system, to demonstrate the feasibility of carrying out large-scale attacks against social authentication with minimal effort on behalf of an attacker. Our experimental evaluation has shown that widely available face recognition software and services can be effectively utilized to break social authentication tests with high accuracy. Overall we argue that Facebook should reconsider its threat model and re-evaluate the security measures taken against it.

Acknowledgements

We thank the anonymous reviewers for their valuable comments and Alessandro Frossi for his support. This paper was supported in part by the FP7 project SysSec funded by the EU Commission under grant agreement no 257007, the Marie Curie Reintegration Grant project PASS, the ForToo Project of the Directorate General Home Affairs, and ONR MURI N00014-07-1-0907. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of ONR or the US Government.

9. REFERENCES

[1] A. Acquisti, R. Gross, and F. Stutzman. Faces of Facebook: How the largest real ID database in the world came to be. BlackHat USA, 2011, http://www.heinz.cmu.edu/~acquisti/face-recognition-study-FAQ/acquisti-faces-BLACKHAT-draft.pdf.

[2] E. Athanasopoulos and S. Antonatos. Enhanced CAPTCHAs: Using animation to tell humans and computers apart. In *Proceedings of the 10th IFIP Open Conference on Communications and Multimedia Security*. Springer, 2006.

[3] M. Balduzzi, C. Platzer, T. Holz, E. Kirda, D. Balzarotti, and C. Kruegel. Abusing social networks for automated user profiling. In *Proceedings of the 13th International Conference on Recent Advances in Intrusion Detection*. Springer, 2010.

[4] L. Bilge, T. Strufe, D. Balzarotti, and E. Kirda. All your contacts are belong to us: automated identity theft attacks on social networks. In *Proceedings of the 18th International Conference on World Wide Web*. ACM, 2009.

[5] Y. Boshmaf, I. Muslukhov, K. Beznosov, and M. Ripeanu. The socialbot network: when bots socialize for fame and money. In *Proceedings of the Annual Computer Security Applications Conference*. ACM, 2011.

[6] E. Bursztein, S. Bethard, C. Fabry, J. C. Mitchell, and D. Jurafsky. How good are humans at solving CAPTCHAs? A large scale evaluation. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*. IEEE, 2010.

[7] E. Bursztein, M. Martin, and J. C. Mitchell. Text-based CAPTCHA strengths and weaknesses. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*. ACM, 2011.

[8] M. Dantone, L. Bossard, T. Quack, and L. V. Gool. Augmented faces. In *Proceedings of the 13th IEEE International Workshop on Mobile Vision*. IEEE, 2011.

[9] R. Dey, Z. Jelveh, and K. Ross. Facebook users have become much more private: A large-scale study. In *Proceedings of the 4th IEEE International Workshop on Security and Social Networking*. IEEE, 2012.

[10] R. Dhamija, J. D. Tygar, and M. Hearst. Why phishing works. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*. ACM, 2006.

[11] R. Dunbar. *Grooming, Gossip, and the Evolution of Language*. Harvard University Press, 1998.

[12] H. Gao, J. Hu, C. Wilson, Z. Li, Y. Chen, and B. Y. Zhao. Detecting and characterizing social spam campaigns. In *Proceedings of the 10th Annual Conference on Internet Measurement*. ACM, 2010.

[13] C. Herley. The plight of the targeted attacker in a world of scale. In *Proceedings of the Ninth Workshop on the Economics of Information Security*, 2010.

[14] D. Irani, M. Balduzzi, D. Balzarotti, E. Kirda, and C. Pu. Reverse social engineering attacks in online social networks. In *Proceedings of the 8th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2011.

[15] D. Jacoby. Facebook Security Phishing Attack In The Wild. Retrieved on January 2012 from http://www.securelist.com/en/blog/208193325/Facebook_Security_Phishing_Attack_In_The_Wild.

[16] M. Jakobsson and J. Ratkiewicz. Designing ethical phishing experiments: A study of (ROT13) rOnl query features. In *Proceedings of the 15th International Conference on World Wide Web*. ACM, 2006.

[17] H. Kim, J. Tang, and R. Anderson. Social authentication: harder than it looks. In *Proceedings of the 2012 Cryptography and Data Security conference*. Springer, 2012.

[18] M. Madejski, M. Johnson, and S. M. Bellovin. A study of privacy settings errors in an online social network. In *Proceedings of the 4th IEEE International Workshop on Security and Social Networking*. IEEE, 2012.

[19] F. Nagle and L. Singh. Can friends be trusted? Exploring privacy in online social networks. In *Proceedings of the 2009 International Conference on Advances in Social Network Analysis and Mining*. IEEE, 2009.

[20] I. Polakis, G. Kontaxis, S. Antonatos, E. Gessiou, T. Petsas, and E. P. Markatos. Using social networks to harvest email addresses. In *Proceedings of the 9th Annual ACM Workshop on Privacy in the Electronic Society*. ACM, 2010.

[21] A. Rabkin. Personal knowledge questions for fallback authentication: Security questions in the era of Facebook. In *Proceedings of the 4th Symposium on Usable Privacy and Security*. ACM, 2008.

[22] A. Shulman. The underground credentials market. *Computer Fraud & Security*, (3), 2010.

[23] J. Staddon and A. Swerdlow. Public vs. publicized: content use trends and privacy expectations. In *Proceedings of the 6th USENIX Conference on Hot Topics in Security*. USENIX, 2011.

[24] B. E. Ur and V. Ganapathy. Evaluating attack amplification in online social networks. In *Proceedings of the 2009 Web 2.0 Security and Privacy Workshop*.

Enabling Private Conversations on Twitter

Indrajeet Singh[1], Michael Butkiewicz[1], Harsha V. Madhyastha[1],
Srikanth V. Krishnamurthy[1], Sateesh Addepalli[2]
[1] University of California, Riverside [2] Cisco Systems

ABSTRACT

User privacy has been an increasingly growing concern in online social networks (OSNs). While most OSNs today provide some form of privacy controls so that their users can protect their shared content from other users, these controls are typically not sufficiently expressive and/or do not provide fine-grained protection of information. In this paper, we consider the introduction of a new privacy control—group messaging on Twitter, with users having fine-grained control over who can see their messages. Specifically, we demonstrate that such a privacy control can be offered to users of Twitter *today* without having to wait for Twitter to make changes to its system. We do so by designing and implementing `Twitsper`, a wrapper around Twitter that enables private group communication among existing Twitter users while preserving Twitter's commercial interests. Our design preserves the privacy of group information (i.e., who communicates with whom) both from the `Twitsper` server as well as from undesired `Twitsper` users. Furthermore, our evaluation shows that our implementation of `Twitsper` imposes minimal server-side bandwidth requirements and incurs low client-side energy consumption. Our `Twitsper` client for Android-based devices has been downloaded by over 1000 users and its utility has been noted by several media articles.

1. INTRODUCTION

OSNs have gained immense popularity in the last few years since they allow users to easily share information with their contacts and to even discover others of similar interests based on information they share. However, not all shared content is meant to be public; users often need to ensure that the information they share is accessible to only a select group of people. Though legal frameworks can help limit with whom OSN providers can share user data, users are at the mercy of controls provided by the OSN to protect the content they share from other users. In the absence of effective controls, users concerned about the privacy of their information are likely to connect with fewer users, share less information, or even avoid joining OSNs altogether.

Previous proposals to address these privacy concerns on *existing OSNs* either (a) jeopardize the commercial interests of OSN providers [31, 23] if these solutions are widely adopted and thus, are likely to be disallowed, or (b) require users, who are currently accustomed to free access to OSNs, to pay for improved privacy [26, 42, 3]. On the other hand, though *new OSNs* have been developed with privacy explicitly in mind [16, 7], these OSNs have seen limited adoption because users are virtually "locked in" to OSNs on which they have already invested significant time and energy to build social relationships. Consequently, users have, in many cases today, raised privacy-related concerns in the media and organizations such as the EFF and FTC have tried to coerce OSNs to make changes. Though OSNs have introduced new privacy controls in response to these concerns (e.g., Facebook friend lists, Facebook groups, Google+ circles), such controls do not provide sufficiently fine-grained protection.

In light of this, we consider the privacy shortcomings on Twitter, one of the most popular OSNs today [17]. Twitter offers two kinds of privacy controls to users—a user can either share a message with all of her followers or with one of her followers; there is no way for a user on Twitter to post a *tweet* such that it is visible to only a subset of her followers. In this paper, we fill this gap by providing fine-grained controls to Twitter users, enabling them to conduct private *group communication*. Importantly, we provide this fine-grained privacy control to Twitter users by implementing a wrapper that builds on Twitter's existing API, and hence, users do not have to wait for Twitter to make any changes to its service.

As our main contribution, we design and implement `Twitsper`, a wrapper around Twitter that provides the option of private group communication for users, without requiring them to migrate to a new OSN. Unlike other solutions for group communication on Twitter [8, 19, 20], `Twitsper` ensures that Twitter's commercial interests are preserved and that users do not need to trust `Twitsper` with any private information. Further, in contrast to private group communication on other OSNs (e.g., Facebook, Google+), in which a reply/comment on information shared with a select group is typically visible to all recipients of the original posting, `Twitsper` strictly enforces privacy requirements as per a user's social connections (all messages posted by a user are visible only to the user's followers).

When designing `Twitsper`, we considered various choices for facilitating the controls that we desire; surprisingly, a relatively simple approach emerged as the best fit for fulfilling our objectives. Thus, our `Twitsper` implementation is based on this simple design which combines a Twitter client (that retains much of the control logic) with a server that maintains minimal state. Importantly, we ensure that no privately shared content is revealed to the `Twitsper` server, and furthermore, the privacy of group memberships is also preserved from both the `Twitsper` server and from other undesired users. Our evaluation demonstrates that

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

this simple design does achieve the best trade-offs between several factors such as backward compatibility, availability, client-side energy consumption, and server-side resource requirements.

Overall, our implementation of `Twitsper` is proof that users can be empowered with fine-grained privacy controls on existing OSNs, without waiting for OSN providers to make changes to their platform. Our client-side implementation of `Twitsper` for Android phones has been downloaded by over 1000 users and several articles in the media have acknowledged its utility in improving privacy and reducing information overload on Twitter.

2. RELATED WORK

Characterizing privacy leakage in OSNs: Krishnamurthy and Willis characterize the information that users reveal on OSNs [34] and how this information leaks [35] to other entities on the web (such as social application providers and advertising agencies). Our thesis is that legal measures are necessary to ensure that OSN providers do not leak user information to third-parties. However, it is not in the commercial interests of OSN providers to support systems that hide information from them. Therefore, we focus on enabling users to protect their information from other *undesired* users, rather than from OSN providers.

Privacy controls offered by OSNs: Google+ and Facebook permit any user to share content with a *circle* or *friend list* comprising a subset of the user's friends. However, anyone who comments on the shared content has no control; the comment will be visible to all those with whom the original content was shared. Even worse, on Facebook, if Alice comments on a friend Bob's post, Bob's post becomes visible to Alice's friend Charlie even if Bob had originally not shared the post with Charlie. Facebook also enables users to form groups; any information shared with a group is not visible to users outside the group. However, a member of the group has to necessarily share content with all other members of a group, even if some of them are not her friends. Twitter, on the other hand, enables any user to restrict sharing of her messages either to only all of her followers (by setting her account to *private* mode) or to exactly one of her followers (by means of a *Direct Message*), but not to a proper subset. We extend Twitter's privacy model to permit private *group communication*, ensuring that the privacy of a user's reply to a message shared with a group is in keeping with the user's social connections.

Distributed social networks: Several proposals to improve user privacy on OSNs have focused on de-centralizing OSNs (e.g., Vis-a-Vis [42], Confidant [37], DECENT [33], Polaris [45], and PeerSoN [26]). These systems require a user to store her data in the cloud or on her own or her friends' personal devices, thus removing the need for the user to trust a central OSN provider. However, users have put in tremendous effort in building their social connections on today's OSNs [4, 17], and rebuilding these connections on a new OSN is not easy. Thus, unlike these prior efforts, we build a backward-compatible privacy wrapper on Twitter.

Improving privacy in existing OSNs: With Lockr [44], the OSN hosting a user's content is unaware of with whom a user is sharing content; Lockr instead manages content sharing. Other systems allow users to share encrypted content, either by posting the encrypted content directly on OSNs [31, 23, 24] or via out-of-band servers [13]. Users can share the decryption keys with a select subset of their connections (friends). Hummingbird [28] is a variant of Twitter in which the OSN supports the posting of encrypted content in such a manner that preserves user privacy. Narayanan et al. [39] ensure users can keep the location information that they divulge on OSNs private via private proximity testing. All of these techniques either prevent OSN providers from interpreting user content, or hide users' social connections from OSNs. Since neither is in the commercial interests of OSN providers, these solutions are not sustainable if widely adopted. In contrast, we respect the interests of OSN providers while exporting privacy controls to users.

Category	%
Consider privacy a concern	77
Would like to control who sees information they post	70
Declined follower requests owing to privacy concerns	50

Table 1: Results of survey about privacy shortcomings on Twitter.

Group communication: Like `Twitsper`, listserv [30] enables communication between groups of users. However, unlike with `Twitsper`, group communications on listserv lack a social structure and listserv was never designed with privacy in mind. Prior implementations of group messaging on Twitter, such as Twitter Groups [20], GroupTweet [8], and Tweetworks [19], have either not focused on privacy—they require users to trust them with their private information—or require users to join groups outside their existing social relationships on Twitter. Similar to `Twitsper`, a recent workshop paper [43] advocated the use of a wrapper that offers private group communication on Twitter, but unlike `Twitsper`, they ignored the leakage of private information, such as the sizes of conversation groups, to the server maintained by the wrapper.

3. MOTIVATING USER SURVEY

While privacy concerns with OSNs have received significant coverage [34, 35], the media has mostly focused on leakage of user information on OSNs to third-parties such as application providers and advertising agencies. Our motivation is the need for a more basic version of privacy on OSNs—protecting content shared by a user from other users on the OSN, which has began to receive some attention [11].

To gauge the perceived need amongst users for this form of privacy, we conducted an IRB approved user study across 78 users of Twitter [1]. Our survey questioned the participants about the need they see for privacy on Twitter, the measures they have taken to protect their privacy, and the controls they would like to see introduced to improve privacy. Table 1 summarizes the survey results. More than three-fourths of the survey participants are concerned about the privacy of the information they post on Twitter, and an almost equal fraction would like to have better control over who sees their content. Further, rather tellingly, half the survey takers have at least once rejected requests to connect on Twitter in order to protect their privacy. These numbers motivate the necessity of enabling users on Twitter to privately exchange messages with a subset of their followers, specifically allowing them to choose which subset to share a message with on a per-message basis.

4. DESIGN OBJECTIVES

Given the need for enabling private group messaging on Twitter, we next design `Twitsper` to provide fine-grained privacy controls to Twitter users. Our over-arching objective in developing `Twitsper` is to offer these controls to users without having to wait for Twitter to make any changes to their service. Our design for `Twitsper` is guided by three primary goals.

Backward compatible: Rather than developing a new OSN designed with better user controls in mind (e.g., proposals for distributed OSNs [42, 26, 3]), we want our solution to be compatible with Twitter. This goal stems from the fact that Twitter already has

[1]Participant details removed for anonymity reasons

Proposal	Backward Compatible	Preserves Commercial Interests	No Added Trust Required
Distributed OSNs	×	×	✓
Encryption	✓	×	✓
Separating content providers from social connections	✓	×	×
Existing systems for group messaging on Twitter	✓	✓	×
Twitsper	✓	✓	✓

Table 2: Comparison of Twitsper with previous proposals for improving user privacy on OSNs.

an extremely large user base—over 100 million active users [21]. Since the value of a network grows quadratically with the growth in the number of users on it (the network effect [36]), Twitter users have huge value locked in to the service. To extract equal value from an alternate social network, users will not only need to re-add all of their social connections, but will further require all of their social contacts to also shift to the new service. Therefore, we seek to provide better privacy controls to users by developing a wrapper around Twitter, eliminating the burden on users of migrating to a new OSN and thus maximizing the chances of widespread adoption of Twitsper.

Preserves commercial interests: A key requirement for Twitsper is that it should not be detrimental to the commercial interests of Twitter. For example, though a user can exchange encrypted messages on Twitter to ensure that she shares her content only with those with whom she shares the encryption keys, this prevents Twitter from interpreting the content hosted on its service. Since Twitter is a commercial for-profit entity and offers its service for free, it is essential that Twitter be able to interpret content shared by its users. Twitter needs this information for several purposes: to show users relevant advertisements, to recommend applications of interest to the user, and to suggest others of similar interest with whom the user can connect. Though revealing user-contributed content to Twitter opens the possibility of this data leaking to third-parties (either with or without the knowledge of the provider), user content can be insured against such leakage via legal frameworks (e.g., enforcement of privacy policies [22]) or via information flow control [46]. On the other hand, protecting a user's content from other users requires enabling the user with better controls—our focus in building Twitsper.

No added trust: In attempting to give users better controls without waiting for Twitter to change, we want to ensure that users do not have another entity to trust in Twitsper; users already have to trust Twitter with their information. Increasing the number of entities that users need to trust is likely to deter adoption since users would fear the potentially greater opportunity for their information to leak to third-parties. Therefore, we seek to ensure that users do not need to share with Twitsper's servers any information they want to protect, such as their content or their login credentials. Tools such as TaintDroid [29] can be used to verify that Twitsper's client application does not leak such information to Twitsper's servers. We design Twitsper for the setting where Twitsper's servers are not malicious by nature, but are inquisitive listeners; this attacker model is similar to that used in prior work (e.g., [40]).

Table 2 compares our proposal with previous solutions for improving user privacy on OSNs. Unlike proposals for distributed OSNs, Twitsper enables users to reuse their social connections on Twitter, and unlike calls for exchange of encrypted content, we respect Twitter's commercial interests. Moreover, we introduce user controls via Twitsper without adding another entity for users to trust, unlike proposals such as Lockr [44], which call for the separation of social connections from content providers. Lastly, in contrast to prior implementations of group messaging on Twitter such as GroupTweet [8], Tweetworks [19], and Twitter Groups [20], we ensure that Twitter is privy to private conversations but Twitsper is not.

API call	Function
PrivSend(msg, group)	Send *msg* to all users specified in *group*
isPriv?(msg)	Determine if *msg* is a private message
PrivReply(msg, orig_msg)	Send *msg* to all of the user's followers who received *orig_msg*

Table 3: Twitsper's API beyond normal Twitter functionality.

5. Twitsper DESIGN

Next, we present an overview of Twitsper's design. We consider various architectural alternatives and discuss the pros and cons with each. Our design objectives guide the choice of the architecture that presents the best trade-offs. As mentioned earlier, surprisingly, a fairly simple approach seems to yield the best trade-off and is thus, used as the basic building block in Twitsper.

Basic definitions: First, we define a few terms related to Twitter and briefly explain the Twitter ecosphere.

- ***Tweet*:** A tweet is the basic mode of communication on Twitter. When a user posts a tweet, that message is posted on the user's Twitter page (i.e., http://twitter.com/username), and is seen on the timeline of everyone following the user.
- ***Direct Message*:** A direct message is a one-to-one private tweet from one user to a specific second user, and is possible only if the latter follows the former.
- @ ***Reply*:** A user can use a @*reply* message to reply to another user's tweet; this message will also appear on the timeline of anyone following both users.
- ***Twitter page*:** Every user's Twitter page(http://twitter.com/username) contains all tweets and @*reply* messages posted by the user. By default, this page is visible to anyone, even those not registered on Twitter. If a user sets her Twitter account to be private, all messages on her page are visible to any of the users following her account.
- ***Timeline*:** A user's timeline is the aggregation of all tweets, direct messages, and @*reply* messages (sorted in chronological order) visible to that user. In addition to her timeline, note that a user can view *any* tweet or @*reply* message posted by any user that she follows by visiting that user's Twitter page.
- ***List*:** Twitter allows every user to create lists—groups of Twitter users selected by the user. Lists can either be public and world viewable, or private and viewable to the user alone.
- ***Whisper*:** Twitsper's private messaging primitive to allow a user to contact any subset of followers

Twitter associates every tweet, Direct Message, user, and list with a unique ID.

Interface: Our primary goal is to extend Twitter's privacy model. In addition to sharing messages with all followers (tweet) or precisely one follower (Direct Message), we seek to enable users to privately share messages with a non-empty proper subset of their followers. To do so, we extend Twitter's API with the additional functionality shown in Table 3. We present the algorithmic representations of these API calls later.

First, the *PrivSend* API call allows users to post *private* messages that can be seen by one or more members in the user's network,

who are *specifically* chosen to be the recipients of such a message. However, simply enabling a message to be shared with a group of users is insufficient. To enable richer communication, it is necessary that the recipients of a message (shared with a group) be able to reply back to the group. In the case of discussions that need not be kept private, a user may choose to make her reply public so that others with similar interests can discover her. However, when Nina responds to a private message from Jack, it is unlikely that Nina will wish to share her reply with all the original target recipients of Jack's message since many of them may be "unconnected" to her. Nina will likely choose to instead restrict the visibility of her reply to those among the recipients of the original message whom she has approved as her followers. Therefore, the *PrivReply* API call enables replies to private messages, while preserving social connections currently established on Twitter via follower-followee relationships. Finally, the *isPriv?* API call is necessary to determine if a received message is one to which a user can reply with *PrivReply*. Hereafter, we refer to the messages exchanged with the *PrivSend* and *PrivReply* calls as whispers.

It is important to note that, since our goal is to build a wrapper around Twitter, rather than build a new OSN with these privacy controls, this extended API has to build upon Twitter's existing API for exchanging messages. Though Twitter's API may evolve over time, we rely here on simple API calls—to post a tweet to all followers and to post a Direct Message to a particular follower—that are unlikely to be pruned from Twitter's API. Also note that, in some cases, multiple rounds of replies to private messages can result in the lack of context for some messages for some recipients, since all recipients of the original whisper may not be connected with each other. In the trade-off between privacy and ensuring context, we choose the former in designing `Twitsper`.

Architectural choices: Next, we discuss various architectural possibilities that we considered for `Twitsper`'s design, to support the interface described above. While it may be easy for Twitter to extend their interface to support private group messaging, we note that Twitter has not yet done so in spite of the need for this amongst its users. Therefore, our focus is in designing `Twitsper` to offer this privacy control to users without having to wait for Twitter to make any changes.

Using a supporting server: The simplest architecture that one can consider for `Twitsper` is to have clients send a whisper to a group of users (represented by a list on Twitter) by sending a Direct Message to each of those users. To enable replies, when a client sends a whisper, it can send to the supporting server the identifiers of the Direct Messages and the ID of the Twitter list which contains the recipients. Thus, a user can query this supporting server to check if a received Direct Message corresponds to a whisper and to obtain the ID of the associated Twitter list. When the user chooses to reply to a whisper, the user's client can retrieve the Twitter list containing the recipients of the original whisper, locally compute the intersection between those recipients and the user's followers, and then send Direct Messages to all those in the intersection.

If the supporting server is unavailable, users can continue to use Twitter as before, except that the metadata necessary to execute the *isPriv?* and *PrivReply* API calls cannot be retrieved from the server. However, the client software can be modified to allow a recipient to obtain relevant mappings (ID of the list of recipients of a whisper) from the original sender. Another option is to have the client embed the ID of the list associated with a whisper in every Direct Message sent out as part of a whisper. However, given Twitter's 140 character limit per Direct Message, this can be a significant imposition, reducing the permissible length of the message content.

This design places much of the onus on the client and may result in significant energy consumption for the typical use case of Twitter access from smartphones. On the flip side, in this architecture, the content posted by a user is not exposed to the supporting server, i.e., privacy of user content from `Twitsper`'s server is preserved. The server is simply a facilitator of group communications across a private group and only maintains metadata related to whispers (we discuss later in Section 6 how we protect the privacy of this metadata as well from the supporting server). Further, Twitter is able to see users' postings and thus its commercial interests are protected. We note that the alternative of the client sending messages to the supporting server for retransmission to the recipients is not an option, since this would require users to trust the supporting server with the content of their messages.

This design however does have some shortcomings. Twitter lacks sufficient context to recognize that the set of Direct Messages shared to send a whisper constitute a single message rather than a local trending topic. Similarly, Twitter cannot link replies with the original message, since all of this state is now maintained at the supporting server.

Posting encrypted content: To address the shortcoming in the previous architecture of being unable to link replies to the original whispers, in our next candidate architecture, we consider clients posting a whisper just as they would a public message (tweet) but encrypt it with a group key which is only shared with a select group of users (who are the intended recipients of the message). This reduces the privacy problem to a key exchange problem for group communications. An out-of-band key exchange is possible.

However, since only intended recipients can decrypt a tweet, Twitter's commercial interests are compromised. Furthermore, filtering of encrypted postings not intended for them is necessary at the recipient's side; if not, a user's Twitter client will display indecipherable noise from these postings. In other words, the approach is not backward compatible with Twitter. Note here that if these issues are resolved, e.g., by sharing encryption keys with Twitter, encryption can be used with any of the other architectural choices considered here to enhance privacy.

Using community pages to support anonymity: Alternatively, one may try to achieve anonymity and privacy by obfuscation. Clients post tweets to a obfuscation server, which in turn re-posts messages on behalf of users to a common "community" account on Twitter. Except for the server, no one else is aware of which message maps to which user. When a user queries the obfuscation server for her timeline, the server returns a timeline that consists of messages from her original timeline augmented with messages meant for that user from the "community" page. The obfuscation prevents the exposure of private messages to undesired users. Since the "community" page is hosted on Twitter, the shortcoming of the encryption-based architecture is readily addressed—Twitter has access to all information unlike in the case of encryption. An approach similar to this was explored in [41].

However, this architecture has several drawbacks. First, Twitter cannot associate messages with specific users; this precludes Twitter from profiling users for targeted advertisements and such. Second, all users need to trust the obfuscation server with the contents of their messages. Finally, since the architecture is likely to heavily load the server (due to the scale), the viability of the design in practice becomes questionable. When the server is unavailable, no private messages can be sent or received.

Using dual accounts: In our last candidate architecture, every user maintains two accounts. The first is the user's existing account, and a second private account (with no followers or followees) is used for sending whispers. When Alice wishes to send a whisper

Design	Twitter's interests preserved	No added trust	Easily scales	Same text size	Always available	Linkable to orig message
Supporting server	✓	✓	✓	✓	✓	×
Embed lists	✓	✓	✓	×	✓	×
Encryption	×	✓	✓	✓	×	✓
Community pages	×	×	×	✓	×	×
Dual accounts (No longer possible)	×	✓	✓	✓	✓	✓

Figure 1: Comparison of architectural choices.

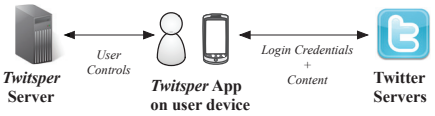


Figure 2: System architecture using supporting server.

to Bob and Charlie, she posts an @*reply* message from her private account to Bob's and Charlie's private accounts. Since Alice's private account has no followers, these @*reply* messages are visible to no users other than to the intended recipients. However, as of mid-2009, Twitter discontinued the "capability" of @*reply* messages between disconnected users after concluding that less than 1% of the users found this feature useful and that it contributes to spam messages [14]. Thus, @*reply* messages posted from these disconnected private accounts will not be visible to intended recipients. Other problems with this architectural choice are that Twitter is unable to associate private messages with the normal accounts of users and responding to private messages is a challenge.

Figure 1 summarizes the comparison of the various architectural choices with respect to our design goals. While no solution satisfies all desirable properties, we see that the use of a supporting server presents the best trade-off in terms of simplicity and satisfying our goals. Therefore, we choose this to be the architectural choice for implementing `Twitsper`, as shown in Figure 2.

While the basic structure of the architecture is simple as discussed above, there exist certain challenges in making the server and other undesired users oblivious to the specifics of a group conversation. We discuss these issues and our approaches for handling them in the following section.

6. PROTECTING PRIVACY

With the supporting server architecture, users do not directly send content to the `Twitsper` server. However, there is metadata that is provided to the server in order to support group conversations —the mapping of Direct Message IDs to list IDs. This metadata could reveal the identities of the members that belong to a private conversation or the group size, and a user may desire to keep such information private. Hence, we incorporate several features that hides this information both from the `Twitsper` server and other undesired users.

Threat model: The components of `Twitsper` are a) Twitter itself, b) user devices, c) the `Twitsper` server, and d) the channel connecting these entities.

We trust Twitter not to leak a person's private information and this has been the premise of our work. We assume that a user's personal device does not compromise a user's privacy; this problem is orthogonal to our work. Thus, the two potential sources of leakage are the `Twitsper` server and the channel. Note here that the `Twitsper` server is the only new addition to the pre-existing Twitter architecture. As discussed earlier, in our supporting server architecture, private content is always posted to Twitter's servers, thus ensuring that this content is not leaked due to the `Twitsper` server. The threat is then the leakage of the metadata associated with private content that may be exposed to the server. Since we administer the `Twitsper` server, we assume that the server will not modify or delete metadata stored on it. Therefore, we focus instead on ensuring that the manner in which metadata is shared with and stored on the `Twitsper` server does not reveal private information either to the server or to undesired users (those not involved in private conversations).

In light of this, we seek to ensure that the following security properties hold:

- An undesired user should not be able to infer which of his/her friends are involved in ongoing private conversations.
- The server should not infer the memberships in ongoing conversations, or determine the size of a private group.

We wish to point out here that if the supporting server has no access to user information (which if made available can compromise the privacy of a user by revealing information such as the number of private conversations that the user is involved in), it cannot authenticate the veracity of whisper postings. We recognize that this exposes the `Twitsper` server to a possible DoS attack wherein fraudulent information could be sent to the server. We defer the exploration of defenses against such attacks on the server for future work, and focus here on protecting user privacy from the server.

Use of certificates to avoid over the channel modifications: The `Twitsper` server has an SSL certificate which validates the authenticity of the server. Thus, a secure HTTPS channel can be established with the server, precluding the possibility of over the channel modifications (as with man in the middle attacks).

Protection from undesired users: A curious user who is not privy to a private conversation may wish to trick the `Twitsper` server into disclosing if one of his friends has initiated a private conversation. To do so, the user may try to guess the message IDs associated with whispers posted by the friend, e.g., based on recent tweets posted by that friend. Note that a whisper results in a set of Direct Messages being posted to Twitter, each of which has an associated message ID.

First, we seek to understand if it is easy for a user to carry out such an attack. Towards this, we perform the following experiments wherein we use three accounts (say) Alice, Bob and Charlie. In our first experiment, Alice sends 50 Direct Messages to Charlie. In our second experiment, Alice sends a Direct Message to Charlie, and immediately thereafter Bob follows by sending a Direct Message to Charlie; we repeat this sequence 50 times. In our final experiment, Alice sends a Direct Message to Charlie and follows that message with a tweet, whereupon Bob does the same. Again, we repeat this sequence 50 times. We observe that while the ID space of tweets and Direct Messages grows monotonically (across both), the gap between the IDs in any pair of posts (sent in quick succession) was *at least* 10^7. We observe no visible pattern using which a user can guess the ID for a Direct Message posted by a friend based on either a recent tweet or Direct Message posted by that friend. While this experiment does indicate that it is hard for an undesired user to query the `Twitsper` server and obtain information with regards to specific private conversations, it does not completely rule out the possibility. Thus, we incorporate the following into our design.

Recall that an initiator of a private conversation sends Direct Messages to a private group, and then seeks to create a mapping on the supporting server between the identifiers for those messages

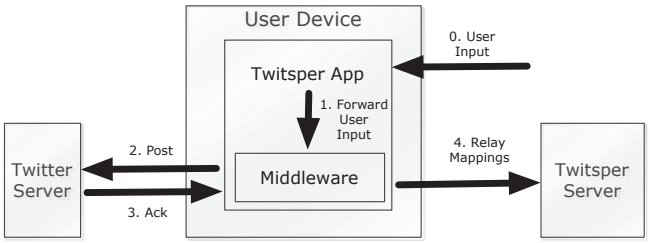
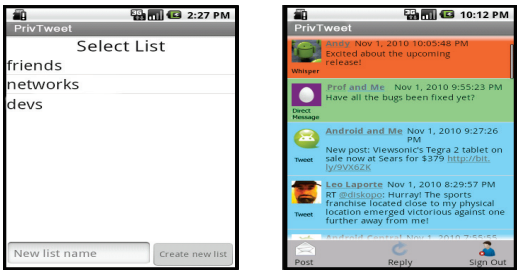


Figure 3: Steps for posting a whisper



(a) List selection (b) A user's timeline

Figure 4: `Twitsper` on Android OS

and the recipient list. Instead of storing this message ID to list ID mapping on the `Twitsper` server simply as $(whisperID, listID)$ tuples, where $whisperID$ is the message identifier assigned by Twitter, we replace the first component in this tuple with the SHA-512 hash value of $(whisperID|userID|text)$. Here, $userID$ corresponds to a receiver of the whisper and $text$ corresponds to the actual content in the message. This way of storing the mappings on the server has two benefits. First, since the hash function is non-invertible, the server cannot infer the identity of the user involved (the $text$ input to the hash function is only known to the group members and thus, not available to the server). Second, even if an undesired user guesses the IDs of the posted messages, he cannot retrieve the desired mapping, again because he does not know the $text$ provided as input to the hash.

Hiding the entries in a list: The list identifiers included in the mappings stored at the `Twitsper` server can however reveal the participants of private conversations to the server. To hide this information, we encrypt the list ID stored in any tuple with a group key. Clearly, the group key should be available to all of the participants themselves but not to the server. Thus, we have all recipients derive a group key K_g from the content of the received Direct Message, which is not exposed to the `Twitsper` server. Since a user may be involved in multiple groups, the private conversation with which a particular received Direct Message is associated may not always be apparent. Therefore, we associate a new group key K_g with every whisper rather than with every conversation. The key K_g for a particular whisper is a function of the associated text and the sender of the whisper encrypts the list ID with K_g before posting the associated mappings to the `Twitsper` server. Finally, though this can impact the availability of metadata, to keep storage costs at the `Twitsper` server low, we purge entries after a pre-specified time interval (days).

Alternatively, we could use a one-to-many or many-to-many stateless broadcast encryption scheme [32, 38, 27, 25], which ensures that re-keying is infrequent and that many possible subsets can be generated with little computational effort. At this point, we did not see any direct advantage of using such approaches over simply deriving the group key for a conversation from the content of the initial Direct Message in that conversation.

Note that, in the rare case where a user has a single list on Twitter, anyone who knows that the user is using `Twitsper` can infer the set of users with whom the user is having private conversations. In practice, we expect that users will conduct private conversations with different groups at different times, and thus maintain multiple lists on Twitter.

Preventing the inference of group sizes: Even though list IDs are now encrypted, the `Twitsper` server can infer the sizes of private groups simply by counting the number of tuples with the same encrypted list identifier. Recall that the list ID is associated with a hash value that is unique to each intended group participant; thus, if there are K participants, there would be K entries corresponding to the same list. Alternatively, it can simply count the number of tuples written by a single client (the initiator) via its HTTPS connection within a short time frame.

To ensure that the $listID$ in its encrypted form cannot be directly used (via counting) to infer the group size, we store entries of the form $enc_{K_g}(listID|hash(listID)|whisperID)$. The $whisperID$ corresponds to the Direct Message sent to a specific receiver, and thus, each entry now has a unique encrypted list ID associated with it; the `Twitsper` server cannot infer group sizes simply by counting tuples with the same second component. It is easy to see that when the entries are sent to users, the client program can decrypt the content and extract the $listID$.

To preclude the server from inferring the group size by counting the number of tuples written by a client within a short time span, we take the following approach. First, note that simply having clients write dummy tuples to the server does not suffice. The server can infer which tuples are spurious by noting the tuples that are never queried. Thus, we associate each entry with a counter value n which can vary from 1 to M, where M is a random value chosen uniquely for each recipient (note that in many cases $M = 1$). We then modify the first and second components of every tuple to be $hash(n|whisperID|userID|text)$ and $enc_{K_g}(n|M|listID|hash(listID)|whisperID)$. For each recipient (say Bob), Alice creates M entries, M being specific to Bob. Of these, as may be evident, $M - 1$ entries correspond to dummy entries. When Bob queries the server for the first time (with $hash(1|whisperID|userID|text)$), he retrieves the value of M and now knows how many spurious entries are stored for him. His client software then sends $M - 1$ additional requests to retrieve the spurious entries.

Our design has several other desirable security properties, that we discuss briefly here.

- **Preventing leakage of the browsing habits of users:** Since the user ID is never directly revealed to the supporting server, the browsing habits or Twitter access patterns of users are held confidential from the server.
- **CCA security:** Our encryption scheme is based on AES (Advanced encryption standard) [5] which ensures CCA (chosen cipher text attack) security. Thus, even with the rather predictable and simple counters used, the list IDs cannot be reverted.
- **Forward and backward secrecy:** Since a new group key is generated per whisper message, even if someone guesses or uncovers the key for the metadata for a specific message, it does not uncover past or future messages both in the same, or in different conversations. This ensures both forward and backward secrecy.

API Call 1 PrivSend(msg,listID)

1: SALT ← First 8 bytes of SHA-512(msg)
2: PASS ← msg concatenated with sender's ID
3: K_g ← PBKDF2(PASS, SALT)
4: **for** each User U in group $listID$ **do**
5: $msgID \leftarrow messageID$ returned by Twitter on successful post
6: M ← select a random number
7: $Entry_a$ ← SHA-512(1|$msgID$|U|msg)
8: $Entry_b \leftarrow encrypt_{K_g}(1|M|listID|hash(listID)|msgID)$
9: EntryList ← add ($Entry_a$, $Entry_b$)
10: **for** $i \in [2, M]$ **do**
11: $Dummy_a^i$ ← SHA-512(i|$msgID$|U|msg)
12: $Dummy_b^i \leftarrow encrypt_{K_g}(i|M|listID|hash(listID)|msgID)$
13: EntryList ← add ($Dummy_a^i$,$Dummy_b^i$)
14: **end for**
15: **end for**
16: **for** each (a,b) in EntryList **do**
17: send (a,b) to `Twitsper` server
18: **end for**

API Call 2 isPriv?(msg)

1: $msgID$ ← Twitter ID for msg
2: $Entry_a$ ← SHA-512(1|$msgID$| self's ID |msg)
3: $response$ ← query `Twitsper` server for $Entry_a$
4: **if** $response \neq null$ **then**
5: SALT ← First 8 bytes of SHA-512(msg)
6: PASS ← msg concatenated with sender's ID
7: K_g ← PBKDF2(PASS, SALT)
8: Decrypt $response$ using K_g and cache embedded $listID$ with $msgID$ for future replies
9: M ← extracted number for spurious queries
10: **for** $i \in [2, M]$ **do**
11: $Dummy_a^i$ ← SHA-512(i|$msgID$| self's ID |msg)
12: $response$ ← query `Twitsper` server for $Dummy_a^i$
13: **end for**
14: **return** TRUE
15: **else**
16: **return** FALSE
17: **end if**

API Call 3 PrivReply(msg,orig_msg)

1: **if** ID for orig_msg is not in cache **then**
2: Reply with a direct message
3: **return**
4: **end if**
5: $listID$ ← mapping for orig_msg's ID in cache
6: $group$ ← group specified by the list ∩ user's followers
7: PrivSend(msg,$group$)

Collision of hash entries: Lastly, since things are indexed by the results of a hash function, the collisions of the hash values might seem to be an issue. The secure hash standard [15] states that for a 512 bit hash function (as in our implementation) we need a work factor of approximately 2^{256} entries to produce a collision which we believe leads to a minuscule probability of experiencing collisions. Thus, we ignore hash collisions for now.

7. IMPLEMENTATION

In this section, we describe our implementation of the `Twitsper` client and server. Given the popularity of mobile Twitter clients, we implement our client on the Android OS [1, 2].

Generic implementation details. Normal tweets (public) and Direct Messages are sent with the `Twitsper` client as with any other Twitter client today. We implement whispers using Direct Messages as described before. Recall that direct messaging is a one-to-one messaging primitive provided by Twitter. Mappings from Direct Messages to whispers are maintained on our `Twitsper` server. Instead of describing each API call separately, our description captures their inter-dependencies.

`Twitsper`'s whisper messages are always sent to a group of selected users. The client handles group creation by creating a list of users on Twitter. This list can either be public (its group members are viewable by any user of Twitter) or private for viewing only by its creator.

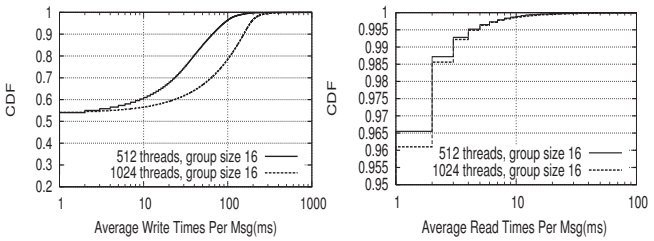
Instantiation of `Twitsper` API: Figure 3 shows the flow of information involved in posting a whisper. The `Twitsper` client at the sender first creates a 256 bit AES key from the content to be shared (msg) using the password-based key derivation function (PBKDF2) from PKCS#5 [10]. The input to PBKDF2 is the message text (msg) concatenated with the user ID of the sender. $SALT$ is a random number generated from the content string; in our implementation we simply use the first 8 bytes of the hash value SHA-512(msg). At the end of these steps, the sender has generated the group key (K_g) for the communication (API Call 1; Lines 1–3). The client then sends a Direct Message via Twitter to each group member, whereupon Twitter returns the message IDs for each recipient (API Call 1; Line 5).

The `Twitsper` client then creates metadata tuples that will enable recipients of the whisper to map Direct Messages to the corresponding list ID (API Call 1; Lines 6–9). Note here that the client also picks a random number M for every recipient and creates $M - 1$ dummy metadata entries on the `Twitsper` server (API Call 1; Lines 10–14) as discussed before. All of these metadata tuples are finally transmitted to the `Twitsper` server (API Call 1; Lines 16–18). As discussed earlier, in order to associate a whisper with the correct list, new metadata is created for every Direct Message sent and K_g is newly generated for every posted whisper.

When the `Twitsper` client program at a recipient receives a Direct Message, it queries the `Twitsper` server to check whether the message is a whisper or a standard Direct Message (API Call 2). To do so, it first computes the SHA-512 hash from the content in the Direct message and its own user ID (API Call 2; Line 2). If the server finds a match for the query string, it returns the corresponding tuple to the recipient client program; else it sends a null response. If an appropriate (non-null) response is received from the server, the `Twitsper` client of the recipient extracts the list ID embedded in the tuple. To decrypt the metadata entry, the client generates the group key K_g using the text in the received Direct Message (msg) and the sender's ID (API Call 2; Lines 5–8). The client also extracts the embedded value of M and sends $M - 1$ additional requests for the spurious entries added for this particular recipient (API Call 2; Lines 9–13).

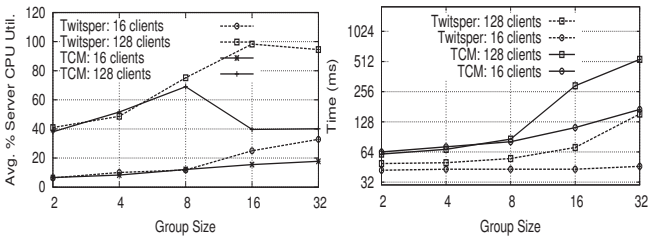
A key feature of our system is that since whispers are sent as Direct Messages, whispers can still be received and viewed by legacy users of Twitter who have not adopted `Twitsper`; such users cannot however reply to whispers (API Call 3). `Twitsper` allows a whisper recipient to reply not only to the sender, but also to a *subset* of the original group (specified by the retrieved list) receiving the whisper. This subset is simply the intersection of the original group and the *followers* of the responding user (API Call 3; Line 6). Thus, it respects the social relations established by users.

Finally, we point out that with the list ID corresponding to the group, the client can retrieve the user IDs on that list from Twitter if the original whisper sender has made the list public. If the list is private, the recipient's response can only be received by the original sender. In the future, we plan to permit `Twitsper` users to modify the list associated with a particular whisper in order to enable inclusion of new users in the private group communication or removal of recipients of the original whisper from future replies; this can be easily done by adding/removing entries on Twitter lists.



(a) DB write time (b) DB read time

Figure 5: Database performance



(a) CPU utilization (b) Service time per client

Figure 6: Server Metrics

Server implementation details: Our server is equipped with an Intel quad-core Nehalem processor, 24 GB of RAM, and one 7200 RPM 1 TB hard disk drive. The `Twitsper` server is implemented as a multi-threaded Java program. The main thread accepts incoming connections and assigns a worker thread, chosen from a thread pool, to service each valid API call. The server stores whisper mappings in a MySQL database. In order to ensure that writing to the database does not become a bottleneck we have multiple connections to the database; we observed that without this, the server performance was affected. These connections are used by worker threads in a round-robin schedule. Note that our server does not store any personal information or credentials of any user. The flow of information in case of a tweet (public) or a Direct Message remains unchanged. Only in the case of a whisper does the use of our system become necessary. The contents of a whisper are never sent to our server; only encrypted metadata is sent as discussed earlier. This ensures that the server can never "overhear" conversations between users or derive user-specific information unless it has either a user's password, which, with `Twitsper`, is never transmitted.

Client implementation details: Our client was written for Android OS v1.6 and was tested on the Android emulator as well as on three types of Android phones (Android G1 dev, Motorola Droid X, and HTC Hero). We use the freely available twitter4j package to access the Twitter API. The client is also multi-threaded and separates the UI (user-interface) thread from the processing, the network, and disk I/O threads. This ensures a seamless experience to the user without causing the screen to "freeze" when the client performs disk or network I/O. We profiled the power consumption of our implementation to identify inefficiencies and iteratively improved the relevant code. These iterative refinements helped us decrease the dependence on the network by caching frequently retrieved user profile images, while maintaining a thread pool rather than the fork and forget model adopted by most open source implementations of other Twitter clients, so as to not over-commit resources.

When the `Twitsper` server is unavailable, we cache whisper mappings on the client and piggyback this data with future interactions with the server. On the other hand, recipients of whispers interpret them as Direct Messages and cannot reply back to the group until the server is again reachable. In future versions of `Twitsper`, we will enable recipients to directly query the client of the original sender if `Twitsper`'s server is unavailable.

We color code tweets, Direct Messages and whispers, while maintaining a simple and interactive UI. Example screen shots from our `Twitsper` client are shown in Figure 4. Our client application is freely available on the Android market, and to date, our `Twitsper` Android application has been downloaded by over 1000 users.

8. EVALUATION

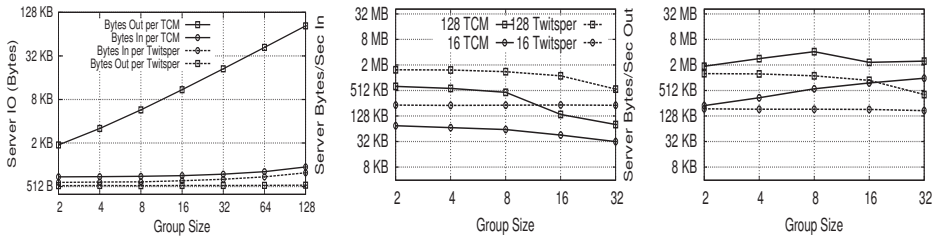
Next we present our evaluation of `Twitsper`. For the purposes of benchmarking, we also implement a version of `Twitsper` wherein a client posts a whisper by transmitting the message to the `Twitsper` server, which in turn posts Direct Messages to all the recipients on the client's behalf. Though, as previously acknowledged, this design clearly violates our design goal of users not having to trust `Twitsper`'s server, we use this *thin client* model (TCM) (we refer to our default implementation as the *fat client* model or `Twitsper` itself) as a benchmark to compare against. One primary motivation for using TCM as a point of comparison is that it can reduce the power consumption on phones (since battery drainage is a key issue on these devices). We also compare `Twitsper`'s energy consumption on a smartphone with that of a popular Twitter client to demonstrate its energy thriftiness.

Server-side results: First, we stress test our server by increasing the rate of connections it has to handle. In this experiment, we use one or more clients to establish connections and send dummy metadata to our server. All clients and the server were on the same local network and thus, network bandwidth was not the constraining factor. We monitored CPU utilization, disk I/O, and network bandwidth with Ganglia [6] and iostat to detect bottlenecks. We vary the target group size of whispers as well as the number of simultaneous connections to the server.

Disk. In Figure 5b, we plot the time taken by each thread to read information relevant to a message from the database (we preloaded the database with 10 million entries to emulate server state after widespread adoption); Figure 5a depicts the CDFs of the write times to the database. We see that as the number of clients increase, so do the database write times, but not the read times. Thus, as the system scales, the bottleneck is likely going to be the I/O for writing to the disk.

CPU. Next, we compare the server performance of TCM and `Twitsper`. We will refer to the version of the server which works in tandem with `Twitsper`, and handles only whisper metadata, as the `Twitsper` server. The TCM server must, in addition, handle the actual sending of whispers to their recipients. It is to be expected that the overhead of the TCM server would increase the computational power needed to service each client. Figures 6a and 6b show the average CPU utilization and user service time, respectively, for each server version. We see in Figure 6a that the `Twitsper` server has a higher CPU utilization than the TCM server. This is because the TCM server spends more idle time (Figure 6b) while servicing each client since it needs to wait on communications with Twitter. So even though more CPU resources are being spent per client with the TCM server, the average CPU utilization is lower.

Another interesting feature noted from these graphs is that certain increases in group size cause the server to more than double its service time. These sharp increases in service time in Figure 6b have corresponding drops in CPU utilization in Figure 6a. This is due to our server's disk writes being the throughput bottleneck. Since in each test we either double the number of client connections or the group size, we would expect a CPU bottleneck to manifest itself with drastic service time increases (of $\approx 200\%$). Instead, the



(a) Bytes in/out per client (b) Incoming bandwidth (c) Outgoing bandwidth

Figure 7: Network activity on server; same legend on (b) and (c)

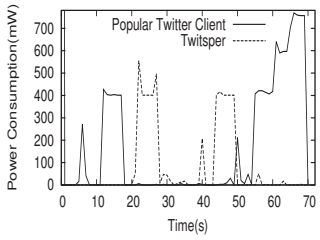


Figure 8: Comparison of power consumption

data points to a disk write bottleneck where the client must wait for an acknowledgment of the server database's successful write. We verify with iostat that our hard drive is used at 100% utilization during these periods. We are currently investigating the effect of adding more disks.

Network. Figure 7a shows the number of bytes in and out with the TCM and `Twitsper` servers for a single client connection. Each line in Figure 7a represents a single client sending one whisper message to a group size which is varied (x-axis). We see that increasing the group size does not cause a large increase in the received bytes as compared to the case with only 2 group members. This illustrates that the overhead increase with recipient group size (which causes either the receipt of more message IDs with the `Twitsper` server or the receipt of more recipient user IDs with the TCM server) is very minor when compared to the resources consumed by the SSL connection between the client and the server. The only additional overhead with the TCM server is the transfer of the actual whisper messages from the client; this manifests as the constant offset between these two curves. Since the `Twitsper` server has to only send a confirmation to the user that its whisper meta data was received correctly, the bytes out is independent of the recipient group size (all meta data corresponding to a whisper is sent as a single atomic block). In contrast, the burden of having to send whispers to each recipient (as a separate Direct Message) is on the TCM server. Increasing group size (x-axis) increases the number of Direct Messages sent to Twitter and this quickly results in an overshoot of the single client SSL connection overhead.

Figures 7b and 7c show the bandwidth consumed at the server as the number of bytes in and out per second. In Figure 7b, we see that the `Twitsper` server does not experience a reduction in transmission rate until it hits 128 clients and a group size of 16. At this point, we hit a disk bottleneck in writing client message metadata to our database. For the TCM server, we see a rate reduction even in the 16 clients case as we increase the group size; this is due to the latency incurred in the message exchange with Twitter. We hit a similar hard disk bottleneck at 128 concurrent client connections with the TCM server, as similar metadata needs to be stored with both server setups.

Comparing `Twitsper` and TCM clients: While `Twitsper` offers higher CPU utilization as well as lower bandwidth requirements, the energy (power∗time) consumption at the client is a key factor in ensuring adoption of the service. To evaluate its client side energy performance, we measure *the amount of energy* needed to make a single post with `Twitsper` to Twitter and to send a message to our server. We also use the PowerTutor [12] application to measure the power consumed at the client. We made 100 posts back to back and measure the average energy consumed.

Figure 10 compares TCM and `Twitsper` based on the energy consumed on a phone. The figure shows the energy consumption per day on an Android phone, for an average Twitter user who sends 10 messages per day and has 200 followers [9]. Our experiments suggest that the best implementation depends on the fraction of a

Interface	`Twitsper`	Other
LCD	13325	10127
CPU	755	1281
3G	4812	8232

Figure 9: Total energy consumption (mJ)

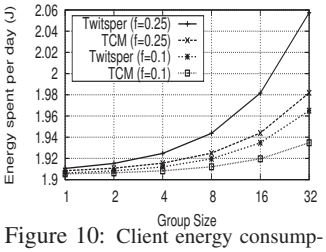


Figure 10: Client energy consumption

user's messages that are private (denoted by f) and the typical size of a list to which private messages are posted. The energy consumption with `Twitsper` is significantly greater than that with the TCM client when f is large or the group sizes are big. However, since we expect private postings to constitute a small fraction of all information sharing and that such communication will typically be restricted to small groups, energy consumption overhead with `Twitsper` is minimal. Even in the scenarios where client-side energy consumption increases, the energy consumed is still within reason, e.g., the energy consumed per client across various scenarios is within the range of 1.9 J to 2.5 J, which is less than 0.005% of the energy capacity of typical batteries (10 KJ, as shown in [12]). Further, as we show next, the majority of the energy consumed in practice is by the user's interaction with the phone's display, whereas the energy we consider here is only that required to simply send messages, and does not include displaying and drawing graphics on the screen.

Comparison with another popular Twitter client: We next compare the power consumption of `Twitsper` with that of a popular Twitter client (TweetCaster[18]), which supports the default privacy options on Twitter. We begin the test after both clients had been initialized and had run for 15 seconds. We then send a message from each of the clients and refresh the home screen; there was at least one update to the home screen. As seen from the traces of the power consumed in Figure 8, `Twitsper`'s power consumption is comparable. This shows that `Twitsper` only imposes energy requirements on the mobile device that are comparable to other Twitter clients. We observe that there is no noticeable loss in performance since both clients were made to carry out the same tasks functionally.

In the above test, even though the screen was kept on for as little a time as possible (less than 10% of the total time) the LCD accounted for close to 50% of the aggregate energy consumed, as seen from Figure 9. Referring the reader back to Figure 10, we see that as the group size increases there is only a marginal increase in the energy consumption associated with the sending of messages. Even if 25% of the messages are whispers and the average group size is 32 (which we believe is quite large), the energy consumed only increases from 1.92 J (for a single tweet) to 2.05 J—an increase of less than 15%; given that the LCD power consumption dominates, this is not a significant energy cost.

9. CONCLUSIONS

Today, for users locked in to hugely popular OSNs, the primary hope for improved privacy controls is to coerce OSN providers via the media or via organizations such as EFF and FTC. In this paper, to achieve privacy without explicit OSN support, we design and implement `Twitsper` to enable fine-grained private group messaging on Twitter, while ensuring that Twitter's commercial interests are preserved. By building `Twitsper` as a wrapper around Twitter, we show that it is possible to offer better privacy controls on existing OSNs without waiting for the OSN provider to do so.

Next, we plan to implement fine-grained privacy controls on other OSNs such as Facebook and Google+ as well, using a similar approach of building on the API exported by the OSN. Given the warm feedback received by `Twitsper`, we hope that the adoption of `Twitsper` and its follow-ons for other OSNs will persuade OSN providers themselves to offer fine-grained privacy controls to their users.

References

[1] Android operating system. http://www.android.com/.

[2] Comscore: Android is now highest-selling smartphone OS. http://bit.ly/euR4Yb.

[3] DiSo project. http://diso-project.org/.

[4] Facebook traffic reaches nearly 375 million monthly active users worldwide, led by us. http://bit.ly/c0Z3UQ.

[5] Fips 197, adavanced encryption standard. 1.usa.gov/8Y4V6U.

[6] Ganglia. http://ganglia.sourceforge.net/.

[7] Google Plus numbers belie social struggles. http://bit.ly/pPIwDr.

[8] Grouptweet. http://www.grouptweet.com/.

[9] New data on Twitter's users and engagement. http://bit.ly/cu8P2s.

[10] PKCS 5: Password-based cryptography specification version 2.0. http://tools.ietf.org/html/rfc2898.

[11] Please rob me. http://www.pleaserobme.com/.

[12] Powertutor. http://bit.ly/hVaXh1.

[13] Priv(ate)ly. http://priv.ly/.

[14] Retweet this if you want non-followers replies fixed. http://bit.ly/YwLYw.

[15] Secure hash standard. 1.usa.gov/cISXx3.

[16] Social networks offer a way to narrow the field of friends. http://nyti.ms/j7dOsC.

[17] Tweet this milestone: Twitter passes MySpace. http://on.wsj.com/dc25gK.

[18] Tweetcaster. http://tweetcaster.com/.

[19] Tweetworks. http://www.tweetworks.com.

[20] Twitter Groups! http://jazzychad.net/twgroups/.

[21] Twitter reveals it has 100m active users. bit.ly/nJoRuk.

[22] Twitter suspends twidroyd & UberTwitter over privacy claims. http://bit.ly/hRcZlw.

[23] R. Baden, A. Bender, N. Spring, B. Bhattacharjee, and D. Starin. Persona: An online social network with user-defined privacy. In *SIGCOMM*, 2009.

[24] F. Beato, M. Kohlweiss, and K. Wouters. *Scramble! Your Social Network Data*. 2011.

[25] D. Boneh and M. Hamburg. Generalized identity based and broadcast encryption schemes. In *ASIACRYPT*, 2008.

[26] S. Buchegger and A. Datta. A case for P2P infrastructure for social networks- opportunities and challenges. In *WONS*, 2009.

[27] Y. Dodis and N. Fazio. Public-key broadcast encryption for stateless receivers. In *ACM Digital Rights Management*, 2002.

[28] G. T. Emiliano De Cristofaro, Claudio Soriente and A. Williams. Hummingbird: Privacy at the time of twitter. Cryptology ePrint Archive, Report 2011/640, 2011. bit.ly/SYBEzK.

[29] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, 2010.

[30] D. A. Grier and M. Campbell. A social history of bitnet and listserv, 1985-1991. *IEEE Annals of the History of Computing*, 2000.

[31] S. Guha, K. Tang, and P. Francis. NOYB: Privacy in online social networks. In *WOSN*, 2008.

[32] J. Y. Hwang, D. H. Lee, and J. Lim. Generic transformation for scalable broadcast encryption scheme. In *CRYPTO*, 2005.

[33] S. Jahid, S. Nilizadeh, P. Mittal, N. Borisov, and A. Kapadia. DECENT: A decentralized architecture for enforcing privacy in online social networks. In *IEEE SESOC*, 2012.

[34] B. Krishnamurthy and C. Willis. Characterizing privacy in online social networks. In *WOSN*, 2008.

[35] B. Krishnamurthy and C. Willis. On the leakage of personally identifiable information via online social networks. In *WOSN*, 2009.

[36] S. J. Liebowitz and S. E. Margolis. Network externality: An uncommon tragedy. *The Journal of Economic Perspectives*, 1994.

[37] D. Liu, A. Shakimov, R. Caceres, A. Varshavsky, and L. P. Cox. Confidant: Protecting OSN Data without Locking it Up. In *Middleware*, 2011.

[38] D. Lubicz and T. Sirvent. Attribute-based broadcast encryption scheme made efficient. In *AFRICACRYPT*, 2008.

[39] A. Narayanan, N. Thiagarajan, M. Lakhani, M. Hamburg, and D. Boneh. Location privacy via private proximity testing. In *NDSS*, 2011.

[40] R. A. Popa, H. Balakrishnan, and A. J. Blumberg. VPriv: Protecting privacy in location-based vehicular services. In *USENIX Security Symposium*, 2009.

[41] M. K. Reiter and A. D. Rubin. Crowds: Anonymity for Web transactions. *ACM TISSEC*, 1998.

[42] A. Shakimov, H. Lim, R. CÃąceres, L. P. Cox, K. Li, D. Liu, and A. Varshavsky. Vis-ã-Vis: Privacy-preserving online social networks via virtual individual servers. In *COMSNETS*, 2011.

[43] I. Singh, M. Butkiewicz, H. V. Madhyastha, S. V. Krishnamurthy, and S. Addepalli. Building a wrapper for fine-grained private group messaging on Twitter. In *HotPETS*, 2012.

[44] A. Tootoonchian, S. Saroiu, Y. Ganjali, and A. Wolman. Lockr: Better privacy for social networks. In *CoNEXT*, 2009.

[45] C. Wilson, T. Steinbauer, G. Wang, A. Sala, H. Zheng, and B. Y. Zhao. Privacy, availability and economics in the Polaris mobile social network. In *HotMobile*, 2011.

[46] N. Zeldovich, S. B. Wickizer, and D. MaziÃĺres. Securing distributed systems with information flow control. In *NSDI*, 2008.

Separation Virtual Machine Monitors

John McDermott, Bruce Montrose, Margery Li, James Kirby, Myong Kang
Center for High-Assurance Computer Systems
Naval Research Laboratory
Washington, DC, US 20375
<firstname>.<lastname>@nrl.navy.mil

ABSTRACT

Separation kernels are the strongest known form of separation for virtual machines. We agree with NSA's Information Assurance Directorate that while separation kernels are stronger than any other alternative, their construction on modern commodity hardware is no longer justifiable. This is because of orthogonal feature creep in modern platform hardware. We introduce the separation VMM as a response to this situation and explain how we prototyped one.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*security kernels,verification*

General Terms

Security

Keywords

virtualization, hypervisor, virtual machine monitor (VMM), open source

1. INTRODUCTION

Use of *virtual machine monitors* (VMMs, using the hardware vendor's term for what are also called *hypervisors*) to share hardware between virtual machines is rapidly increasing. For security, VMMs are an attractive alternative to hardware separation. Conventional VMMs are typically designed to provide secure separation of their exported virtual machines but the strongest known separation for shared execution environments is provided by a *separation kernel* [26, 22]:

> the task of a separation kernel is to create an environment which is indistinguishable from that provided by a physically distributed system: it must appear as if each regime is a separate, isolated machine and that information can only flow from one machine to another along known external communication lines.

(Rushby used the term *regime* to denote the hardware abstraction a separation kernel provides to a VM.) The separation kernel is a concept proven by practical application. We have a great deal of knowledge about how they can be constructed and used. Separation kernels are already available as, for example, the emerging MILS platform [2, 6] and the Green Hills Integrity® VMM [1]. Separation kernel software is small and simple, so it makes a small target that can be completely analyzed and verified mathematically.

Separation kernels represent the upper bound of practical software separation strength, but they are no longer justifiable on commodity hardware. The National Security Agency's Information Assurance Directorate (IAD) discusses this issue in its guidance for separation kernels on commodity workstations [31]. Their essential conclusion is that strict separation kernels are still the strongest known means of separation, but that there is no rationale for building strict separation kernels because of the increasing *orthogonal feature creep* of modern commodity hardware. Orthogonal feature creep makes commodity hardware too complex for separation kernels. The IAD report states

> Ultimately, the problem with commodity desktop platforms comes down to the fact that too many developers and vendors are interdependent. Each organization involved in the creation of a desktop workstation has an economic interest in adding features to distinguish their version from the competition and often needs a particular, unique, and potentially powerful access. This prevents a unified strategy for security from emerging.

For commodity hardware, we need to find and understand the new practical upper bound for separation strength. It is critical to understand that we are not proposing to change the meaning of high robustness. Instead, we want to find a better approach to measuring, building, and using VMMs that have the greatest robustness that is justifiable for commodity hardware.

The first consideration would seem to be how such VMMs might be used but actually this begs the question of what is the practical upper bound of assurance that can be justified. This is the goal of the 𝕏enon project, to investigate this new practical upper bound of assurance for modern commodity hardware.

If we take Rushby's separation kernel concept as an approach rather than a specific architecture, we can make sig-

ACSAC '12, December 3-7, 2012, Orlando, FL
ACM 978-1-4503-1312-4/12/12

nificant progress towards finding and understanding a justifiable upper bound. Lacking full mathematical verification, the alternative will not separate as robustly as a strict separation kernel. However it will provide stronger separation than commercial/open source best practice VMMs, for modern commodity hardware.

As an approach, the separation kernel combines the following concepts:

- partition all computation into VMs,
- isolate the VMs to use a small number of well-understood communication paths, and
- verify the security properties of the VMM to the highest level possible.

If we relax the assurance of the separation kernel from complete mathematical verification to the highest level commensurate with modern commodity hardware, but retain the strict isolation concept, we get a *separation VMM*. A separation VMM will

- run on modern commodity hardware,
- virtualize modern commodity operating systems,
- be smaller and simpler than a conventional VMM,
- use fewer and simpler communication paths, and
- have the highest assurance justifiable for its modern commodity hardware.

2. VMM SECURITY ALTERNATIVES

There are several general approaches to increasing the run-time security of a VMM:

- add an external run-time integrity verification mechanism,
- add an internal run-time integrity verification mechanism,
- add further self-protection mechanisms, to increase tamper resistance,
- reduce the size and complexity of the software, to decrease the number of residual security flaws, or
- use formal methods to decrease the number of residual security flaws.

Various combinations of these measures are possible as well. Research into these approaches has significantly increased our understanding of how VMM security can be improved. Separation kernels represent an extreme combination of the last 2 approaches; separation VMMs are a less extreme combination of the same.

Each approach also brings with it some undesirable impacts and shortcomings. Adding software of any kind typically increases space and time overhead but it also increases the size of the code base, most probably introducing new security flaws. If we accept the motivation for adding the software, viz. security flaws in software are inevitable, then we must accept that an addition itself has flaws. None of the reported VMM security research on adding software discusses measures taken to reduce the residual flaw density of the added software (or flaws in its interaction with the existing VMM software). The added software may also increase the attack surface of the VMM, by providing new ways to access its run-time internals.

Integrity verification mechanisms face additional challenges. All verification mechanisms face challenges of sampling rate and coverage. In modern commodity hardware, small but significant parts of the security state are either difficult to capture or change too rapidly to sample in a reliable way. It is difficult to show that there is no way for an attacker to bypass these kinds of defenses. External verification mechanisms are relatively tamper-resistant, but sample too slowly and cannot observe all of the security state. Internal verification mechanisms are better at sampling but are also relatively easy to tamper with. Integrity verification defenses that are non-trivial are hard to assure. Their security properties are complex, i.e. precise definition of all possible attacks and how the integrity mechanism is sure not to miss any of them.

Augmenting the existing self-protection mechanisms of a VMM is an intuitive response to VMM security. (To save space, we will assume that the reader is already familiar with hardware protection mechanisms such as no-execute, write-protect, and other paging attributes, segment registers, cache-line address space identifiers, and guard pages.) Advanced Micro Device's (AMD's) decision to remove the segment register protection mechanisms from the x86_64 instruction set architecture (ISA) may have been justified by the fact that current operating systems were not using it, but the security results of segment register protection in the Multics project still apply [4, 15]. There is no characteristic of modern operating systems that would rule out use of this strong and efficient mechanism.

This lack of proper segment registers (or alternative compartmentation mechanisms) in the widely-used x86_64 ISA is a strong motivation for adding further self-protection mechanisms to VMMs. These added mechanisms are designed to directly prevent tampering with code in memory, malicious diversion of control flow, and access to unauthorized address spaces. The challenge is to design software mechanisms that both satisfy the three reference monitor properties [1] but also are efficient. New protection mechanisms must also permit typical system programming techniques such as function pointers and self-modifying code.

Reducing the size and complexity of the software not only decreases the number of residual security flaws in a VMM, it also strengthens its reference monitor property of verifiability. There are four fundamental challenges to this approach. First, as the IAD guidance concludes, modern commodity hardware is growing more and more complex, and virtualization of all of this under complexity constraints is hard. Second, modern unmodified guest operating systems are even more complex than the hardware, but a practical VMM must support this complexity. Third, simplicity can introduce significant performance penalties. It can be difficult to design code that is both simple and fast; especially under the restrictions imposed by complex commodity hardware and guest operating systems. Fourth, it can be difficult to report research results on approaches that reduce size or complexity. Peer-reviewed venues prefer novel, powerful, and flexible security features, which tend to be large and

[1] completeness, isolation, and verifiability

complex rather than simple and small. To achieve significant results, approaches that reduce the size or complexity of the VMM, for security, must run on hardware that is arguably a commodity, e.g. x86_64 or ARM, and support commodity operating systems such as the various forms of Microsoft Windows, Apple OS X, or an unmodified standard Linux distribution. A significant result for minimizing the hypervisor must also retain all of the features needed to support cloud computing.

Formal methods applied to small (less than 10,000 source lines of code (SLOC)) separation kernels or embedded software systems can raise security assurance to the highest level. Even successful approaches have placed significant restrictions on standard system programming techniques such as the use of pointers. In addition to the verification size challenge per se, that is, how to verify enough code to assure an actual product, formal methods approaches also face the 4 challenges of size/complexity reduction approaches: complex underlying hardware, support for complex guests, performance penalties, and the lack of significance entailed by toy projects. We agree with the IAD guidance that mathematically verified separation kernels are not justifiable on modern commodity hardware.

3. PROTOTYPING A SEPARATION VMM

We prototyped the separation VMM concept using the Xen open source VMM. Our prototyping is limited to the Xen VMM itself. The 𝕏enon VMM will work with any of the conventional Xen control plane tools such as *xm*, *xl*, or *libvirt*. The prototype is tested continuously against not only Windows and various Linux distributions but also Hadoop, Zookeeper, and Accumulo. This confirms that our extensive refactoring and re-design of the VMM internals has preserved 𝕏enon's support for commodity software.

It is important to understand that our separation VMM prototype is not a security enhancement to Xen, but a separate VMM code base that does not necessarily conform to Xen community practices. The VMM security features that we describe here are neither recommended as changes to Xen, because they do not conform to Xen community goals and practices, nor, for the same reason, are they to be viewed as criticisms of Xen's design.

Our prototype is based on Xen 4, starting with a 4.0.1 basis but then forward porting to a Xen 4.1.2 basis (to measure the difficulty of a forward port). While justifiable assurance is a key question of our work, that assurance must have a reasonable target. So the 𝕏enon prototype not only explores separation VMM construction but also acts as a target for investigation of justifiable assurance.

𝕏enon's code is not meant to be a special branch of the Xen code base. 𝕏enon draws from Xen's code, for selected new features and bug fixes, but separation VMM simplicity and assurance requirements rule out maintaining it as a branch of Xen. We validated this strategy by measuring the effort needed to keep the prototype synchronized with both the fixes and the new features of Xen [21]. The effort of analyzing, translating, vetting (not all bug fixes are applied to 𝕏enon) and applying all of the Xen bug fixes created during a 6 month period was measured at approximately 100 hours for an experienced kernel coder [21]. The effort needed to synchronize new features from one Xen branch to another (4.0.1 to 4.1.2) was measured at approximately 200 hours.

Our changes to the Xen code have resulted in a VMM that is 26% smaller than conventional Xen, in terms of lines of code, with worst-case complexity [36] reduced from 2,450 to 70, a 3500% decrease. We report these values as approximate percentages because they change frequently, but downward over time. Table 1 shows specific values for Xenon change set 341 and its base Xen 4.1.2. Some simplifications also increase the total amount of code in the hypervisor, for example when a complex C function is broken up into many smaller functions.

Directory	Xen	𝕏enon
arch	64006	46520
common	15507	13204
include	3558	1691
crypto	1113	1072
drivers	19287	19256
xsm	6665	0
msm	0	269
Total	110136	82012

Table 1: Size Reduction of 𝕏enon VMM in Non-comment Lines of Code. MSM is the Xenon replacement for XSM.

4. SEPARATION VMM SECURITY

We wanted our separation VMM prototype's enforcement mechanisms to be as verifiable as we could make them. We also wanted the security to be intuitive to use.

A virtual machine monitor and its guest virtual machines constitute a *virtual machine system* [25]. The combined security model for a VM system must define policies that are both consistent and intuitive. The need for consistency in security VM system models and policies is addressed by Rueda et al. [25] so we will not discuss it further here. The need for intuitiveness is less clear but no less important. Intuitiveness is related to *safety* [14]: a complex unintuitive security model and its policies may be described in a way that is theoretically tractable but leaves administrators of real systems unable to easily determine whether the configured policy satisfies their requirements. Ultimately, the policy enforced by a VMM must be easy to understand and to construct, otherwise users will disable or work around it. A complex resource-based policy expressed as rules about low-level platform-specific resources will be relatively difficult to translate into the high-level service agreements needed by cloud or SOA management. If a guest migrates to another VM system, it may need a different set of low-level rules on its new host VMM, because the devices and other peripherals it was using on the source have different names are not present on the destination host. We would like to have a policy that uses high-level rules that can be the same on all VMMs. We would like to have rules that cover equivalent resources, without being expressed in terms of specific resources.

Security in conventional Xen is enforced primarily by its Xen Security Module (XSM) but also by other small pieces of code at key points in the hypervisor. The XSM security model is based on SELinux (in turn based on Flask [29]) which provides flexible fine-grained type enforcement on individual resources. This flexibility comes at a price: the policies require many rules and it is difficult to translate the policy into an intuitive abstraction of what protection

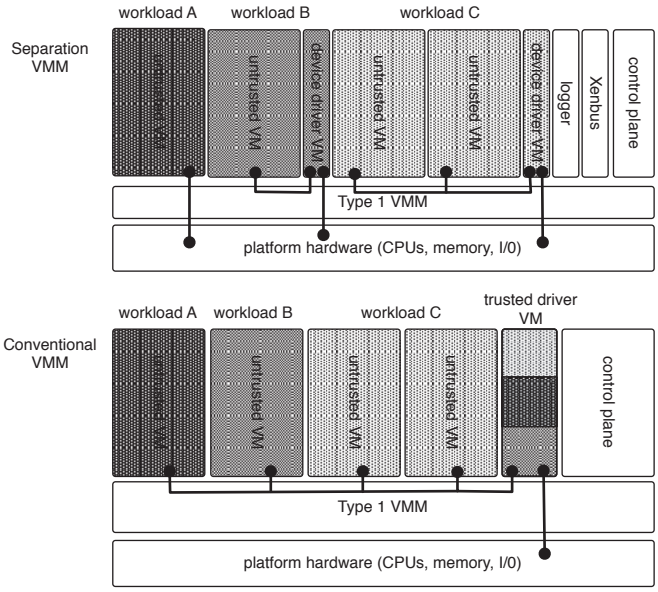


Figure 1: Separation VMM Security Mechanisms Are Simpler Because They Support Simpler Policies

a given policy specifies. The current practice for Xen is to enable security in a passive mode and record the violations, then create rules to address the violations one-by-one. While this approach yields a working policy, it cannot be used to construct a policy that is verifiably a refinement of a previously chosen high-level abstraction.

𝕏enon security policies are separation VMM policies with limited communication between guests. 𝕏enon does not conform to the Xen code base. For these reasons we were able to replace Xen's security implementation with an order of magnitude smaller implementation (269 versus 6665 lines of code) that protects all of the Xen code paths protected by Xen, but also provides additional VMM security features. As shown in Figure 1, 𝕏enon virtual machine system configurations do not include trusted guests, e.g. a trusted device driver domain that serves guests whose information is supposed to be separated.

In the example of Figure 1, there are 3 workloads A, B, and C that are to be separated. While a conventional hypervisor can be configured to this same arrangement, it can also be configured as shown in the lower part of the figure, with a single driver domain that is trusted not to share anything contrary to the configured policy. The 𝕏enon prototype does not support trusted guest configurations and thus its security mechanism is much simpler.

Security in 𝕏enon is enforced entirely by a single module named MSM that replaces both the XSM security module and other code located at various places in the Xen internals. Like Xen, 𝕏enon's MSM enforces a policy described by a specification. The policy specification is written in XML and is compiled into a binary policy that is loaded into the VMM's memory at boot time.

The number of rules in a specification is relatively small. For example, suppose that the workloads A, B, and C shown in the top of Figure 1 each run on a different kind of guest operating system. The workload A's guest is configured for direct pass through of the underlying I/O hardware and provides its own device drivers. The single workload B guest has its own device driver domain that is configured to only access workload B resources, i.e. the device driver VM is untrusted. The 2 workload C guests are supported by a single untrusted driver VM. There are two additional service VMs: one for security logging and another to separate the Xenbus service [8, 33] from the domain 0 control plane. A complete policy for this configuration would require only 11 rules.

An MSM security policy contains rules describing

- domains,
- communication allowed between domains,
- rules for domain labels, and
- hypercalls allowed for each domain.

The XML for the rules includes features for aliases, profiles to collect frequently used sets of rules, domain rules, and domain sets to collect domains into a single named group. In the following, we explain each of these MSM features.

4.1 Domain Policy

A domain defines the virtual hardware environment that a guest operating system runs in, just as Xen does. Unlike Xen, 𝕏enon always assigns the same domain id to the same domain. Security enforcement in Xen is based on an SSID that is generated from a domain's UUID. To simplify from this design, 𝕏enon assigns a fixed domain id that always corresponds to specific domain. MSM then uses the simple domain id as the security identifier. This design simplification is typical of our approach; since 𝕏enon does not have to conform to the Xen code base we are free to implement direct solutions without getting sign off from the entire community.

The following policy fragment shows a single domain rule that defines a security logging domain. The profile defines the hypercalls that a security logging domain may make; the label is a mandatory access control label; and the UUID field is used to map the running VM back to its installed image.

```
<domain
      name="MsmLogger"
      id="5" profile="Logger" label="GUEST"
      uuid="1eee192b-1303-ecd4-da1d-8267fae46a1d"/>
```

To provide greater flexibility during the installation and configuration of a 𝕏enon VM system, the domain policy allows execution of anonymous *unprotected* domains that are not defined in the policy. Unprotected domains are assigned domain ids starting at 10000. MSM will allow creation of VMs in unprotected domains, as long as no protected (i.e. defined in the policy) domains are executing their VMs. This allows an administrator to install, test, or otherwise provision VMs without security enforcement. When the VM system is otherwise ready for operational use, domains can be incrementally added to the security policy and tested as protected domains.

4.2 Communication Policy

MSM enforces communication between guests as a simple (upper triangular) *communication matrix* that defines which domains are allowed to communicate. 𝕏enon's communication policy is a simple all-or-nothing policy that is easy to

understand and verify: any 2 domains in a policy are either allowed to communicate and otherwise share resources, or they are not allowed to share anything. This is consistent with the separation VMM concept of using a small number of communication paths between domains. In the XML policy, communication rules are expressed as connections from domains to sets of domains. If a domain does not appear in any connection, then it can only connect to itself. This is a good example of how the separation kernel principle influences and simplifies the design of a separation VMM: we assume that most domains will not communicate with other domains and thus the default is easily described by the lack of a rule. The other type of communication pattern for a separation VMM is a *driver domain pattern* where a driver domain is allowed to connect to a set of domains that have the same workload or other security attributes.

The following is an example policy based on the top of Figure 1. There are no connection rules for the workload A guest or the logger VM because they do not connect to any other domain except the control plane, and follow the default pattern. The control_plane VM rule says that it is allowed to connect to guests in the special domain set All, which allows it to connect to any domain, even those that do not appear in the policy and are thus otherwise unconnected. The 2 workload driver rules allow the appropriate driver VMs to connect to their respective domains.

```
<connection from="control_plane" to="All" />
<connection from="workload_c_driver" to="workload_c" />
<connection from="workload_b_driver" to="workload_b" />
```

4.3 Label-Based Domain Policy

𝕏enon's MSM provides mandatory access control (MAC) labels for enforcement of coarse-grained per-virtual-machine policies. Labels can be defined and applied to domains and also to the domain's associated disk and network interfaces. MSM labels are not limited to lattice-based information flow policies; they can be used by 𝕏enon to enforce a variety of rules including Chinese Wall conflict sets [7, 27], type enforcement [5, 34], and time-based rules. An example of the latter would be a domain label that restricted a domain to running only during normal working hours.

MAC labels are a key component of simplified security enforcement in the 𝕏enon prototype, but their use is well understood, so we do not discuss them at length. Further discussion of label-based mandatory access control policy in VMMs can be found in Sailer, et al. [27].

4.4 Hypercall Policy

Labels provide a MAC policy in the 𝕏enon prototype, at the granularity of a domain. We gain extra flexibility over domain-level MAC policies through management of individual resources. 𝕏enon and Xen can enforce policies over exactly the same set of resources, but 𝕏enon enforces less fine grained separation of resources than Xen. Instead of the Flask-style per-individual-resource rules of Xen, 𝕏enon enforces resource control on a per-hypercall basis. 𝕏enon's resource policies supplement its label-based domain access control. For example, supplementary rules can be useful when MAC labels are being used for Chinese Wall conflict sets. Two domains that are allowed to execute together according to a MAC Chinese Wall policy can still be separated in other ways, e.g. USB port restrictions, by resource policy rules.

In 𝕏enon each guest is given a profile defining which hypercalls and hypercall subcommands the guest is allowed to make. (Xen hypercalls are organized into primary hypercalls with subcommands in order to reduce the size of the hypercall page [8].) If a guest attempts to make a hypercall that is not in its profile then its request is rejected and logged (see below). 𝕏enon's MSM can be configured with a threshold for the number of security violations allowed by a guest. If a guest attempts more unauthorized hypercalls than the threshold then the guest is terminated by the hypervisor, and the attempt is both reported to the console and permanently recorded in the MSM security log. Since all resources are accessed via hypercalls, checking on a per-hypercall basis provides essentially the same protection. The distinction is that Xen can have two individual resources for a single domain labeled with different security labels. 𝕏enon either allows the resource class to a domain or denies it.

MSM includes hooks that check the hypercall privileges of the guest, before the VMM begins execution of the hypercall. Hypercalls in Xen are implemented as a simple call to an entry in the guest's *hypercall page*, after the guest places all parameters in general purpose registers. Each hypercall page contains a mode-specific piece of trampoline code [8] that bounces the flow of control in a way that is appropriate for the guest's mode of operation, e.g., HVM mode for guests using hardware virtualization. For example, in the case of a guest running in HVM mode on an Intel processor, the guest's hypercall page contains trampoline code that executes a `vmcall` instruction which transfers the flow of control into the VMM to perform the hypercall. In our HVM mode example, the VMM code to execute the hypercall is in C function `hvm_do_hypercall`. The MSM hook is placed in this C function, to check all Intel HVM mode guest hypercalls.

Basing resource management on the VMM "API" hypercalls rather than on individual resources enables a significant reduction in the size and complexity of the security policy specification, as depicted by the diagram in Figure 2. Because a single hypercall controls many resources, the number of rules can be reduced, in comparison to a policy that specifies individual resources directly. The reduction in security enforcement code makes this clear. The drawback to this approach is the need to construct a resource-to-hypercall map, as part of an assurance argument, to show that all resources are protected. Since an assurance argument will need list all resources anyways, the additional map is not a significant assurance burden. Describing policies using per-resource rules will eliminate this map, but cause a combinatorial explosion of rules that is less intuitive.

4.5 Protecting Multicall

The multicall feature is potentially vulnerable to a time-of-check-time-of-use attack against the separation policy. Multicall allows guests to reduce virtualization hypercall overhead by grouping multiple hypercall requests into a single multicall hypercall. A typical use would be grouping multiple requests to update a page table. Multicall requests pass in a variable-length list of guest handles that point to guest memory locations. Each list entry is a data structure corresponding to a single hypercall request. This list is not copied into the VMM's address space, to save time but also to avoid overrun attacks.

The initial trampoline-based security check can only con-

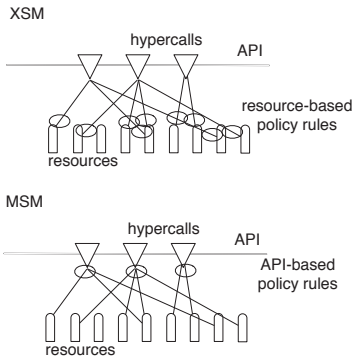


Figure 2: Comparison of resource- and API-based security policy rules.

firm a guest's privilege to perform multicalls; it cannot check the entire list, because of the impact that would have on preserving the Xen-compatible interface of 𝕏enon. Even if the trampoline-based check could confirm every entry on the list, a guest could still perform a time-of-check-time-of-use attack by modifying the list after the multicall was initiated. So the separation policy component of the MSM security module checks each list entry as part of the code that processes the multicall list internally. Each hypercall request is checked just before it is performed.

4.6 Logging

The MSM security logging feature allows the 𝕏enon separation VMM to report its security events to external security management layers or intrusion detection and prevention systems. Conventional Xen's XSM does not provide logs of security activity. This is not unusual, because persistent logging of events of any kind is problematic for VMMs. Excepting a rudimentary console for the local host hardware, a VMM does not have its own file system, high-level network interfaces (for remote persistent storage), or even disk drives per se. Instead, a VMM manages or exports devices such as block or serial devices, so any persistent storage that could be used for sensor data must be implemented by a guest. Conventional Xen's non-security VMM logging is implemented as writes to Xen's local console device.

MSM logging collects security events as fixed-length (currently 512 byte) records held in VMM memory. A single-purpose logging guest is given the privilege of registering with the VMM and subsequently pulling log records from VMM memory. All security logging interaction with the VMM is through a logging-specific hypercall that is controlled through the separation policy, so only authorized guests can manipulate the security logs. A typical configuration would be to have the logger directly export the log records into to a remote host, so that the log data could be incorporated into intrusion detection systems.

𝕏enon uses a special purpose logging guest operating system based on Xen's mini-os. This gives the logging guest a small code base, less than 10,000 lines of source code in its present form, and a correspondingly small attack surface.

4.7 Hypercall Flooding

The 𝕏enon prototype also includes a simple defense against *hypercall flooding attacks*. In a hypercall flooding attack, a malicious guest generates a large number of hypercalls in an attempt to consume VMM resources. The malicious guest makes specious hypercalls but discards the results and does not wait for the VMM response. This is analogous to network attacks based on SYN flooding [3]. The 𝕏enon VMM can be configured to measure and limit the hypercall rate for its guests. Our measurements show a typical rate of 30 hypercalls per millisecond for domain 0, running on either the Xen VMM or 𝕏enon. The 𝕏enon prototype can be configured to limit the hypercall rate in terms of hypercalls per millisecond. The defense is simple and adds no measurable overhead.

VMM.VM	1st Run	2nd Run	3rd Run	Change Set
Xen.alpha	167.02	167.11	167.04	0
Xen.delta	198.19	197.95	197.94	0
Xenon.alpha	167.25	167.13	167.31	294
Xenon.delta	199.00	199.16	199.21	305
Xenon.alpha	167.14	167.06	167.21	343

Table 2: Single-VM in seconds for kcbench. Change Set 0 is the Xen base. VM alpha is a PVops HVM Linux guest; VM delta is a paravirtualized Linux guest.

4.8 Minimizing Overhead

One of the goals of the 𝕏enon project is to reduce the runtime space and time performance overhead of both the security and assurance modifications to conventional Xen. We tested each change for performance, using a variety of commodity hardware, with several benchmark tools. The benchmarks were configured to stress test the prototype VM system while measuring its performance. We have been able to keep the runtime overhead within acceptable limits by backing out any changes that perform poorly. This approach let us simplify the code by over 3000% with no impact (the raw data shows the simplified code actually ran faster, but the difference was in the noise). More information about 𝕏enon performance measurements are presented in [21]. We present a brief table here of single VM data from a kcbench Linux compiler benchmark, as Table 2. The data is for a machine using Hardware Assisted Paging (HAP), with an Intel® S1200BTL base board, a single Intel® Xeon® (E31270) processor (3.20GHz) and 16GiB of memory. The domain 0 kernel was Fedora 3.1.9-1.fc16.x86_64; the same kernel was used for the PVops HVM guests and the paravirtualized guest was installed from the same Fedora 16 ISO image. The examples are for different 𝕏enon change sets, to show how we have been able to maintain performance as the internals are simplified.

An important part of security overhead is the cost of factoring Xen's single large domain 0 into multiple single-purpose security domains. Each one of these domains brings with it some scheduling and context switch overhead. The overhead of the MSM logging domain is so small that we were unable to measure it accurately, that is, the cost of logging is negligible. The data for change set 343 in Table 2 shows the performance change caused by running the security logger; notice that for the 2nd run, the measurement actually shows the VM system to be faster with the logger than without, i.e. the difference is not measurable in this test.

5. JUSTIFIABLE ASSURANCE

Given the separation VMM concept and a working prototype, we need a more concrete notion of highest justifiable assurance for modern commodity hardware. We need to understand how orthogonal feature creep reduces the value of individual components of an assurance argument. The key question is: does orthogonal feature creep reduce the value of a specific measure to the point where it is no longer justified?

We can use the amount of rework required by a change in hardware as the measure of value reduction. The fraction of a work product that must be changed to preserve an assurance argument, with respect to hardware variation, can be viewed as the "cost" of the hardware variation.

Using this perspective, we can identify specific assurance measures that are less sensitive to orthogonal feature creep than complete mathematical verification but still exceed commercial/open source best practice for VMM construction. For the 𝕏enon project, we found the following 6 measures to be justifiable for modern commodity hardware. These measures also increase assurance beyond commercial/open source best practice, Common Criteria EAL4, or both. We do not consider our collection of measures to be definitive for either the concept of highest assurance justifiable for modern commodity hardware or separation VMMs. Instead, they provide a consistent package of assurance measures that can be applied to a separation VMM to give higher assurance than commercial/open source best practice etc. but not be as sensitive to orthogonal feature creep.

5.1 Reduced Features and Size

Commercial/open source best practice does not reduce features (or the size of a product) to obtain greater security. Reducing the features of a software product strongly tends to reduce both its market share and installed base. VMM software also suffers from orthogonal feature creep. So a key element of the separation VMM concept is to push back against this creep by directly reducing the features of a conventional VMM, and then reducing the size and complexity of the remaining software. The reduction is constrained by the requirement for the reduced VMM to virtualize precisely the same guests that the conventional VMM supports. Reducing the features of a product strongly tends to reduce both its attack surface and the total residual flaws present in the product. Reducing product features is easily justified as an assurance measure. It reduces the likelihood that a given hardware variation will impact the VMM and reduces the total amount of rework because the both the product and its assurance argument is smaller. Feature and size reduction is also essential to the separation VMM concept.

Space limitations prevent us from explaining all of the feature and size reductions we applied in constructing the 𝕏enon prototype. We can give a few examples

- support for the Itanium processor: our initial decision was simply to limit the source code to a single instruction set architecture (ISA). We dropped the Itanium because there is a greater variety of commodity hardware for the x86 ISA family,
- support for the 32-bit x86 processor: we removed support for this older member of the x86 ISA family,
- transcendent memory: the design of this feature does not properly scrub memory before re-using it and the feature is not essential,
- supervisor mode kernel: this feature is a mode of VMM execution that supports running a single VM with the same privilege level as the VMM itself. This single supervisor mode kernel VM is the only VM that can run in this mode. This VMM mode is not essential to normal virtualization.

5.2 Reduced Complexity

In this case, we mean reducing complexity to be lower than commercial/open source best practice. Developing simple but correct and efficient code is time consuming; simplifying code beyond commercial/open source best practice not only reduces the likelihood of security weaknesses but also makes it easier to review by automatic or manual means. Simplifying the VMM internals to satisfy CC requirement ADV_INT.2 also increases assurance beyond EAL4.

In our prototype, we simplified code by the application of a small set of patterns. This was only possible because a great deal of the Xen code consistently uses other patterns and it is possible to re-factor the Xen patterns into the Xenon patterns. This refactoring still requires developers analysis; it is probably not practical to use code generation or other machine translation techniques to generate the simplifications. In some cases, e.g. in bringing the complexity of C function `x86_emulate` down from 2,450 to 27 [2], it was necessary to design an intermediate form of the code, to support changing and testing the new code in small increments. The final stage of the re-factored code follows the Xenon simplification patterns, but the mapping is not direct from old to new.

The use of a small set of patterns to re-factor the Xen code is essential to the relatively low cost of keeping pace with the conventional Xen code base. Because of the patterns, an experienced programmer can quickly spot the points where the Xenon code should be changed, when reading a Xen patch.

5.3 Programming Language Subset

Use of a programming language subset exceeds commercial/open source best practice for VMMs. While subsets are commercial best practice for safety critical embedded systems, subsets are specialist practice for general commercial software development. Specialist use of subsets for safety critical systems makes the case that subsets provide greater assurance. Use of a programming language subset exceeds all Common Criteria assurance requirements. For a given language, the use of a carefully designed subset should increase assurance with minimal risk from hardware variation.

For the 𝕏enon project, we chose Hatton's EC– subset [12] of the C programming language as a basis. We had to modify this subset because the EC– subset is designed for ISO C 9899 but Xen is implemented using the GCC compiler in a way that does not conform to the ISO standard. For example, Xen's implementation contains many uses of the VARARGS macros, which breaks the ISO C standard. For 𝕏enon the modified subset uses the EC– rules that are independent of ISO C, and then conforms to the GCC features

[2]The careful reader will have noticed that 27 is less than the value of 70 mentioned earlier. After simplification, function `x86_emulate` is no longer the most complex function in Xenon.

Xen uses to enhance the assurance of the code, e.g. the `-Werror` flag.

5.4 Formal Security Model

Common Criteria EAL4 does not require a formal security model to guide the design and implementation of the security features of a software product. However, using a formal security model (requirement ADV_SPM.1) does increase assurance as it is used in CC EAL's 6 and 7. Because it is very abstract, a formal security model is not likely to require significant (if any) rework because of hardware variation.

Franklin et al. [10] have already developed a formal model for the sHype security mechanism of Xen [27]. They model checked the Xen sHype Chinese Wall model using the Murϕ model checker. (The sHype mechanism was implemented in Xen as the Access Control Module (ACM) but ACM has been replace by NSA's XSM, as described above.)

McDermott and Freitas developed a formal security model specifically for Xenon [19]. This model used the Circus formalism [37] to adapt the independence policies of Roscoe, Woodcock, and Wulf [23, 24] into a state-based framework. This kind of formal security model is especially suitable for practical separation VMMs because it is a definition of security that is preserved by refinement.

5.5 Formal Interface Specification

Common Criteria EAL4 does not require a formal interface specification to guide the design and implementation of the security features of a software product. A formal interface specification (ADV_FSP.6) is used in CC EAL7, so it does increase assurance. It also leverages the assurance provided by a formal security model. An interface specification is much less abstract than a security model, but significantly more abstract than internal design or source code. A formal interface specification is likely to require rework due to hardware variation, but not enough to rule out its use. Orthogonal feature creep is also mitigated by the fact that there is no formal relationship between the interface specification and the underlying design and source code, but only a semiformal one at best. The cost of reworking a semiformal mapping is significantly less than the cost of remapping a formal refinement.

Freitas, McDermott, and Woodcock developed a partial formal interface specification [11] for Xenon. The specification modeled the event channel interface in the Circus formalism and was verified using a combination of tools including the Z / Eves theorem prover and the Community Z Tools (CZT) suite. The specification also demonstrated refinement of the formal security security policy model [19] of McDermott and Freitas.

5.6 Abstract Formal Design Models

While formal verification of the complete design is not justifiable, it is justifiable to verify abstract models of parts of a design. For example, Franklin et al. have model checked the design of the Xen shadow paging mechanism [9], to show that it correctly isolates one guest's memory from another. Formally verifying the abstract design of a VMM subsystem or component is not just an academic exercise, previous research by Franklin et al. has discovered flaws in actual VMMs [10]. Since the design models are abstract, they are less sensitive to changes in hardware.

6. RELATED WORK

There is a rapidly growing body of virtualization security research. We only discuss work that is closely related to our own goal: a separation VMM that has arguably higher security than a conventional VMM. Examples of conventional VMMs would be open source Xen distributions such as Citrix XenServer, VMWare's ESXi, and Microsoft's Hyper-V.

Early IBM work by Sailer et al. [27] developed a mandatory access control framework for Xen called Access Control Module (ACM). Xenon's MSM follows some of the design principles of the IBM ACM framework.

Szefer et al. have prototyped NoHype [32]. NoHype severely reduces the size of the run-time interface between the VMM and the guest virtual machines of a platform, by locking guests to specific processor cores and reducing hypervisor interaction to a stub that handles certain privileged events such as VM exits. This makes it very unlikely that any guest can attack the VMM through its interface.

Ultimately, the NoHype approach is very similar to allocating separate hardware blade servers, from a blade enclosure, to each customer, and is not really virtualization at all, as confirmed by the title of Keller et al. [16]. The design sacrifices some of the benefits of cloud computing on a virtualized infrastructure. There is an added drawback when compare to dedicated blade servers: NoHype still has the software VMM and Linux control plane (domain 0) running on the shared hardware.

HyperSafe [35] adds software self-protection mechanisms to VMMs that run on commodity hardware, to compensate for the hardware inadequacies discussed above. The HyperSafe project prototyped 2 alternative mechanisms: *memory lockdown* based on x86 hardware support for copy-on-write and *restricted pointer indexing*. The memory lockdown feature was constructed using conventional tools, but the restricted pointer indexing is based on the specialized LLVM compiler [18]. Full results have been reported for prototyping these techniques on BitVisor [28]. BitVisor is too small and simple to support cloud computing or similar applications, but instead strongly protects a single guest. Wang and Jiang report partial results for the more complex Xen hypervisor: the memory lockdown technique has been prototyped but no code size or performance results were given [35].

NOVA [30] increases VMM security by layering it into 1) a more privileged component (micro-hypervisor) that directly controls the hardware and manages VM exits, and 2) a user space per-guest VMM that emulates sensitive instructions, handles VM exits, and implements virtual devices. Communication between the privileged layer and the per-guest VMMs is by messages sent from the privileged layer to portals in the per-guest VMMs. For example, if a per-guest VMM executes a sensitive instruction, the per-guest VMM performs a VM exit and the privileged VMM transmits the correct state to a handler at the appropriate port in the per-guest VMM. The extra transitions between layers do impose an overhead compared to conventional virtualization technology; the precise amount varies depending on the hardware/software configuration used. The NOVA prototype currently runs unmodified Linux guests but is actively under development to reduce overhead and support Windows guests.

Zhang et al. [38] have prototyped CloudVisor to use nested virtualization and encryption to protect the guest

VMs, the higher level VMM (a modified Xen VMM), and the control plane VM. The underlying CloudVisor VMM is tiny, about 5K SLOC. CloudVisor does have some performance impact, e.g. worst case I/O intensive applications over 50%. The CloudVisor approach can provide strong security but the threat model precludes cloud providers from inspecting the computing practices of tenant guest machines, to ensure that tenants abide by their good user agreements. For clouds where the provider does not need to enforce any kind of restrictions on tenant behavior, the CloudVisor approach is a good fit.

The Xenon project constructed a Xen based prototype [20] that focused on transforming open source Xen 3 into a Common Criteria defined higher assurance form. Unlike the prototype reported here, the earlier Xenon 3 prototype was a partial prototype that did not change Xen security subsystems. The 𝕏enon prototype has re-factored the entire Xen 4 code base and made extensive changes to the Xen security subsystems.

Current examples of full mathematical verification applied to VMM-like systems included Klein et al. [17] and Heitmeyer et al. [13]. The *L4.verified* project of Klein et al. is targeted at cell phones and similar hardware. The formal verification applied to C source code (i.e. a practical verification) but placed impractical restrictions on the use of pointers (for system software) and did not verify memory management. The definition of security verified was not a information flow security definition, but the take-grant model, which is much simpler than a true information flow model that would be needed by a separation kernel. The embedded device verified by Heitmeyer et al. has much higher assurance and is verified to enforce a true information flow model. It is accurately described as a strict separation kernel but it does not run on commodity hardware.

7. CONCLUSIONS

Our results confirm IAD's position on modern commodity hardware. In parts of Xenon where the code is largely independent of the underlying hardware, a useful separation kernel style policy can be enforced with less than 500 lines of code. In the parts of the VMM that must deal with orthogonal feature creep and legacy hardware features, e.g., the paging code, the reduction cannot be as much. We tried simplifying the paging code but the performance penalty for the simpler code was in excess of 5%, i.e. too much for a single security feature.

We have been successful in reducing the size of a conventional VMM to roughly 3/4 of its original size without losing the ability to virtualize the same operating systems as Xen. Some of this success is due to orthogonal feature creep in the conventional VMM software but some is also due to being focused on a different goal. The Xen community is concerned with security but makes fewer concessions to it when design tradeoffs are considered. The 𝕏enon VMM puts security on par with function and performance.

Our ability to simplify the code exceeded our expectations. At the beginning of the project we did not think it would be possible to actually simplify C function `x86_emulate` by such a large amount and we further did not expect it to be as fast as the original. Our results indicate that using an open source VMM as the basis for a higher assurance separation VMM is practical.

Our performance results for the security logging domain are encouraging. They suggest that it should be possible to refactor Xen's domain 0 into a number of simpler management domains, without harming performance.

Application of formal methods to abstract representations can be useful. Formal models of not only the hypercall interface but also abstractions of key subsystem design (e.g. the shadow paging) and the fundamental security policy can increase the assurance of a VMM, without excessive sensitivity to orthogonal feature creep.

Our future work with 𝕏enon will focus on further size reduction, reducing the size of the domain 0 control plane, and construction of evidence for less-than-full-mathematical-verification. None of this work would have been possible without the high-quality Xen code base.

8. REFERENCES

[1] Green Hills Software INTEGRITY-178B Separation Kernel, comprising: INTEGRITY-178B Real Time Operating System (RTOS), version IN-ICR750-0101-GH01_REL running on Compact PCI card, version CPN 944-2021-021 with PowerPC, version 750cxe. Science International Applications Corporation (SAIC), September 2008.

[2] J. Alves-Foss, W. S. Harrison, P. Oman, and C. Taylor. The MILS architecture for high assurance embedded systems. *International Journal of Embedded Systems*, 2((3/4)), 2006.

[3] R. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems, 2^{nd} ed.* Wiley Publishing, Inc., 2008.

[4] A. Bensoussan, C. Clingen, and R. Daley. The Multics virtual memory: concepts and design. In *Proc. Symposium on Operating Systems Principles (SOSP)*, 1969.

[5] W. E. Bobert and R. Y. Kain. A practical alternative to heirarchical integrity policies. In *Proc. 8th National Computer Security Conference*, Gaithersburg, Maryland, US, 1985.

[6] C. Boettcher, R. DeLong, J. Rushby, and W. Sifre. The MILS component integration approach to secure information sharing. In *27th IEEE/AIAA Digital Avionics Systems Conference*, 2008.

[7] D. Brewer and M. Nash. The Chinese wall security policy. In *Proc. IEEE Symposium on Research in Security and Privacy*, pages 206–214, Oakland, California, US, May 1989.

[8] D. Chisnall. *The Definitive Guide to the Xen Hypervisor*. Prentice-Hall, 2008.

[9] J. Franklin, S. Chaki, A. Datta, J. McCune, and A. Vasudevan. Parametric verification of address space separation. In *Proc. 1st Conference on Principles of Security and Trust (POST)*, Tallin, EE, March 2012.

[10] J. Franklin, S. Chaki, A. Datta, and A. Seshadri. Scalable parametric verification of secure systems: How to verify reference monitors without worrying about data structure size. In *Proc. IEEE Symposium on Security and Privacy*, Oakland, California, US, May 2010.

[11] L. Freitas, J. McDermott, and J. Woodcock. Formal methods for security in the Xenon hypervisor. *International Journal on Software Tools for Technology Transfer (STTT)*, 13(5):463–489, 2011.

[12] L. Hatton. EC– a measurement based safer subset of ISO C suitable for embedded systems development. *Information and Software Technology*, 47(3):181–187, 2005.

[13] C. Heitmeyer, M. Archer, E. Leonard, and J. McLean. Formal specification and verification of data separation in a separation kernel for an embedded system. In *Proc. 13 ACM Conf. on Computer and Communications Security*, Alexandria, Virginia, US, 2006.

[14] T. Jaeger and J. Tidswell. Practical safety in flexible access control models. *ACM Trans. on Information and System Security*, 4(2):158–190, May 2001.

[15] P. Karger and R. Schell. Thirty years later: Lessons from the Multics security evaluation. In *In Proc. Annual Computer Security Applications Conference*, 2002.

[16] E. Keller, J. Szefer, J. Rexford, and R. Lee. Virtualized cloud infrastructure without the virtualization. In *International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society Press, June 2010.

[17] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cook, P.Derrin, D. Elkaduwe, K. Englehardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proc. 22nd ACM Symposium on Operating System Principles*, Big Sky, MT, US, October 2009.

[18] C. Lattner. LLVM: An infrastructure for multi-stage optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, 2002.

[19] J. McDermott and L. Freitas. A formal security policy model for Xenon. In *Proc. Formal Methods in Security Engineering (FMSE '08)*, October 2008.

[20] J. McDermott, J. Kirby, B. Montrose, T. Johnson, and M. Kang. Re-engineering Xen internals for higher-assurance security. *Information Security Technical Report*, 13(1):17–24, 2008.

[21] J. McDermott, B. Montrose, M. Li, J. Kirby, and M. Kang. The Xenon separation VMM: Secure virtualization infrastructure for military clouds. In *Military Communications Conference - MILCOM 2012*, Orlando, FL, US, October 2012.

[22] B. Randell and J. Rushby. Distributed secure systems: Then and now. In *23rd Annual Computer Security Applications Conference (ACSAC)*, Miami, FL, US, December 2007.

[23] A. Roscoe. CSP and determinism in security modelling. In *Proc. IEEE Symposium on Security and Privacy*, Oakland, California, US, May 1995.

[24] A. Roscoe, J. Woodcock, and L. Wulf. Non-interference through nondeterminism. In *Proc. ESORICS*, Brighton, UK, November 1994.

[25] S. Rueda, H. Vijayakumar, and T. Jaeger. Analysis of virtual machine system policies. In *Proc. ACM Symposium on Access Control Models and Technologies (SACMAT)*, Stresa, Italy, June 2009.

[26] J. Rushby. Design and verification of secure systems. *Proc. ACM Symposium on Operating System Principles*, 15:12–21, 1981.

[27] R. Sailer, T. Jaeger, E. Valdez, R. Cáceres, R. Perez, S. Berger, J. Griffin, and L. van Doorn. Building a MAC-Based security architecture for the Xen open-source hypervisor. In *Proc. 21st Annual Computer Security Applications Conference*, Tucson, Arizona, US, December 2005.

[28] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato. BitVisor: a thin hypervisor for enforcing I/O device security. In *Proc. 2009 ACM SIGPLAN/SIGOPS Int. Conf. on Virtual Execution Environments*, pages 121–130, Washington, DC, US, 2009.

[29] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask security architecture: system support for diverse security policies. In *Proceedings of the 8th conference on USENIX Security Symposium - Volume 8*, Washington, DC, US, 1999.

[30] U. Steinberg and B. Kauer. NOVA: a microhypervisor-based secure virtualization architecture. In *Proc. 5th European conference on Computer Systems*, pages 209–222, Paris, FR, 2010.

[31] Systems and N. A. Center. *Separation Kernels on Commodity Workstations*. Information Assurance Directorate, NSA, March 2010.

[32] J. Szefer, E. Keller, R. Lee, and J. Rexford. Eliminating the hypervisor attack surface for a more secure cloud. In *Proc. Computer and Communications Security*, Chicago, IL, US, October 2011. ACM.

[33] C. Takemura and L. Crawford. *The Book of Xen*. No Starch Press, 2010.

[34] K. Walker, D. Sterne, M. L. Badger, M. Petkac, D. Shermann, and K. Oostendorp. Confining root programs with domain and type enforcement (DTE). In *Proc. 6th USENIX UNIX Security Symposium*, San Jose, California, US, July 1996.

[35] Z. Wang and X. Jiang. HyperSafe: a lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proc. 31st IEEE Symposium on Security & Privacy*, Oakland, California, US, May 2010.

[36] A. Watson and T. McCabe. *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric, NIST Special Publication 500-235*. National Institute of Standards and Technology, 1996.

[37] J. Woodcock, A. Cavalcanti, M.-C. Godel, and L. Freitas. Operational semantics of Circus. *Formal aspects of computing*, 2008. in press.

[38] F. Zhang, J. Chen, H. Chen, and B. Zang. CloudVisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proc. 23rd ACM Symp. on Operating Systems Principles (SOSP)*, pages 203–216, Cascais, Portugal, 2011.

Efficient Protection of Kernel Data Structures via Object Partitioning

Abhinav Srivastava*
AT&T Labs-Research
abhinav@research.att.com

Jonathon Giffin
HP Fortify
jgiffin@hp.com

ABSTRACT

Commodity operating system kernels isolate applications via separate memory address spaces provided by virtual memory management hardware. However, kernel memory is unified and mixes core kernel code with driver components of different provenance. Kernel-level malicious software exploits this lack of isolation between the kernel and its modules by illicitly modifying security-critical kernel data structures. In this paper, we design an access control policy and enforcement system that prevents kernel components with low trust from altering security-critical data used by the kernel to manage its own execution. Our policies are at the granularity of kernel variables and structure elements, and they can protect data structures dynamically allocated at runtime. Our hypervisor-based design uses memory page protection bits as part of its policy enforcement. The granularity difference between page-level protection and variable-level policies challenges the system's ability to remain performant. We develop kernel data-layout partitioning and reorganization techniques to maintain kernel performance in the presence of our protections. We show that our system can prevent malicious modifications to security-critical kernel data with small overhead. By offering protection for critical kernel data structures, we can detect unknown kernel-level malware and guarantee that security utilities relying on the integrity of kernel-level state remain accurate.

1. INTRODUCTION

Monolithic operating system kernels contain a collection of code and data attributable to the core kernel and each of its dynamically-loaded components, such as drivers and modules. Widespread software attacks regularly increase their potency by manipulating security critical data in the kernel using a technique called direct kernel object manipulation (DKOM) [39]. Kernel-level malware often uses this technique to hide the existence of malicious processes by eliding task structures for the processes from the kernel's process accounting list. Malicious kernel-level components can hide their own presence by illicitly removing data structures identifying their presence from a kernel-managed list of loaded drivers or modules. They may escalate a process' privileges by overwriting the process' credentials with those for a root or administrative user. Security software relying upon the core kernel's own management information will fail to identify the presence of malicious software. In many attack instances, security software that periodically verifies the consistency of core kernel data structures will be unable to locate any errors.

In this work, we present a system called *Sentry* that creates access control protections for security-critical kernel data. We partition kernel memory into separate regions having different access control policies or restrictions detailing when the code of the core kernel and its loaded components can access sensitive data in a particular protected region. To avoid both direct and indirect attacks from kernel-level malware, our access control enforcement occurs within a hypervisor executing beneath the kernel running in a virtual machine (VM). The hypervisor mediates the execution of instructions attempting to write protected sensitive kernel data. All other accesses to non-sensitive data occur without any mediation. Sentry enforces access control policies that specify what code regions of the kernel are allowed to write what data objects within kernel memory. Notably, Sentry does not rely on white-listing of drivers or modules and offers its protection to security-critical kernel data from all dynamically loaded components.

Our access control protections are general, start from boot, and encompass both statically and dynamically allocated sensitive kernel data objects. Providing access control to static, read-only sensitive data objects is straightforward because they remain unaltered during the execution of operating systems. As evasive attacks often circumvent detection by manipulating heap-managed kernel data objects, we focus primarily on protections for dynamically-allocated sensitive data structures. In our design, the kernel communicates to the hypervisor the need for mediated access to a newly constructed data object at the time that it constructs the object.

Memory access control provides security for critical kernel data at the cost of diminished runtime performance. Our hypervisor-based enforcement mechanism verifies memory accesses at the granularity of high-level language *variables* in the kernel's source code, including individual structure fields contained within a larger aggregate object. We implement mediated access at *page-level* granularity, the only granularity of memory protection offered by commodity hardware. This mismatch leads to inefficiencies in straightforward implementations of memory access mediation. A memory page containing even one security-critical variable would require the entire page's data to be protected. While Sentry can easily distinguish between security-critical and non-critical regions within a single page, it would still introduce performance cost for accesses

*Parts of this work were completed when both authors were at the Georgia Institute of Technology.

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

to all non-critical variables on the page. In the worst case, every access to every kernel data object would be mediated; we expect that the costs would be extremely high.

To address the performance challenge, we present an approach that alters the kernel's data structure design and runtime memory layout. Rather than allowing security-critical and non-critical data to freely intermix on memory pages, we segregate data based on its access policy. We cluster together data having the same access policy on the same memory pages, and separate data with different policies onto different pages. In this design, Sentry will be invoked only for accesses to data requiring mediation, as all non-critical data will be placed on unprotected memory pages.

Critical data protection provides a variety of system and software security techniques with assurances in the integrity of kernel state. Malware that attempts to hide itself or a process by manipulating kernel data will trigger an access violation in Sentry; if the malware chooses not to attempt the alteration, then traditional malware detectors that search for suspicious modules, drivers, and processes will locate the attack. Security software using virtual machine introspection (VMI) [17] to understand the security state of an untrusted guest operating system likewise can rely upon state information in the guest protected by Sentry.

We developed Sentry using the fully-virtualized Linux operating system and the Xen hypervisor [7], which uses hardware virtualization for the guest VM [5]. We tested Sentry with a collection of rootkits that use DKOM to make malicious modifications to kernel data. Sentry successfully prevented all attacks that attempted to alter kernel data illegitimately and allowed only legitimate changes performed by the core kernel. We tested Sentry's performance using both micro and macro benchmarks, and we found that Sentry incurs 0–1% overhead on CPU and network bound processes and less than 4% overhead on mixed workloads. The results indicate that Sentry incurs very low overhead, and its design significantly improves the system's performance when compared with the cost of protecting an unoptimized, unpartitioned kernel (0–16% overhead on CPU and network bound processes and 27% overhead on mixed loads).

We used the Linux operating system due to the availability of its source code. The concepts introduced in this work are equally applicable to closed source operating systems such as Windows. Windows developers can create a partitioned kernel during an OS development cycle to support our protection mechanisms. Although our approach requires small code revisions, similar code-level changes are performed to adopt security mechanisms such as PatchGuard [23], a new driver architecture [2], ASLR [4], and other security techniques [3]. We believe that the code-level change requirement will not be a limiting-factor in the adoption of our work for the long-term improvements to kernels' security.

In this paper, we make the following contributions:

- We create protected memory regions within the unified kernel data space. A kernel can then isolate its security-critical data from kernel components having low trust, creating assurance in the critical state.
- We show how to optimize kernel memory space layout for the protection constraints created by our system. Our layout changes do not impact correctness of kernel execution, but they allow our access control enforcement to operate with higher performance.
- We design and develop a system, *Sentry*, which is capable of protecting both statically and dynamically allocated data structures. We protect dynamically-allocated security-critical kernel objects that are frequently altered by illegitimate, untrusted kernel components because this significantly advances defense capabilities. Sentry detects kernel-level malware that use DKOM to alter security-critical kernel data maliciously.

2. RELATED WORK

Our primary contribution is integrity protection of kernel data. Previous studies have examined aspects of this problem and arrived at solutions different than our own. Petroni et al. [28] proposed a system that detects semantic integrity violations in kernel objects, such as a process task structure reachable when traversing a linked list used by the scheduler but not reachable when traversing the process accounting list. Baliga et al. [6] improved Petroni's work by developing an automated system to generate invariants on kernel data structures. Periodic invariant verification attempts to discover the sort of data manipulation addressed by our work, but it has some limitations overcome by Sentry. These techniques succeed only when invariants can be stated for a data object. This is clearly possible for structures like a process list, as the invariant can compare the reachable nodes along two different traversals: process accounting list and process scheduling list. It is not evident that invariants can be written for other structures, such as the list of loaded kernel modules, which do not offer multiple views. Sentry operates differently: it mediates all attempted data alterations and allows only those invoked by legitimate kernel functionality. While previous approaches detect malicious modifications, Sentry prevents the illegitimate changes of critical data from even occurring.

XFI [15] and BGI [11] provide integrity protection for data (among other protections) via guarded write instructions in software components subject to access control policy constraints. Their use of inlined verification imposes restrictions on software development that may prove difficult to satisfy in actual deployments due to the presence of malicious drivers. For example, they require buy-in from all kernel drivers and modules (including rootkit modules). In contrast, Sentry operates with only cooperation from the core, static kernel; dynamically-loaded components are unconstrained. The designs of XFI and Sentry also highlight differences between inlined monitoring and external protection. XFI guards all computed writes to ensure that no write kills a protected value. When many of these writes target unprotected addresses, performance still degrades. Sentry, in comparison, mediates writes only when they actually attempt modification of protected data. XFI's protections occur at the origin (every write instruction), whereas Sentry's protections occur at the destination (the security-critical data).

Many kernel-level attacks hook a kernel's control flow by overwriting a code pointer stored in a static kernel data table or variable. On the Windows platform, PatchGuard [23] protects read-only structures containing code pointers, such as the system service descriptor table (SSDT) and interrupt descriptor table (IDT). Petroni and Hicks [29] detected illicit control-flow transfers arising from hooked code pointers. These systems protect read-only dispatch tables or known potential hooks. The goals of Sentry are more general—it provides more extensive protection by additionally mediating access to dynamically-allocated data structures.

We partition kernel memory and data objects into protected and unprotected regions. The general concept of partitioning an object into secure and insecure portions has appeared as a solution to other software security and reliability problems. Multics' protection rings created different memory regions having different access permissions [13]. The Pentium architecture created hardware based rings to isolate user-level processes and kernel-level services [18]. Mondrian [42], a fine-grained memory protection system, allows

multiple protection domains to share and export services. Payne et al. [27] split a single security application among multiple VMs and protected the components placed in any untrusted VM. Chong et al. [12] sub-divided web applications to ensure that the resulting placement of code and data are secure and efficient. Like these examples of interface and application partitioning, the goal of Sentry's memory partitioning is to improve kernel security. However, unlike previous systems, the purpose of Sentry's object partitioning is to improve the performance of the protected kernel.

Sentry enforces memory access policies at the hypervisor level and executes the protected operating system within a virtual machine. Wide-ranging security applications use virtualization technology to help build attack-resistant computer systems. Uses include file system protection [26], exfiltration detection [35], malware analysis [14], and other security services [16,20,30,32,36,38]. Our low-level memory access mediation uses memory page protection bits to cause deliberate faults to the hypervisor when kernel code attempts to alter a protected variable. SecVisor [33] protected static kernel code, though it provided no protection for kernel data. Wang et al. [41] protected kernel function pointers by redirecting them to a common secure location. Litty et al. [22] developed a system that neutralizes the ability of rootkits to hide executing binaries from the administrator by leveraging the non-executable bit of memory pages. Xiong et al. [43] designed memory protection based technique that relies on whitelisting of drivers to protect commodity kernels from untrusted extensions. Sentry uses similar manipulation of page protections to control and monitor memory use within a guest OS.

3. OVERVIEW

A running kernel aggregates code and data from the core kernel and from a collection of dynamically-loaded modules and drivers. These different components may engender varying levels of trust in regions of the kernel. We assume that the core kernel implemented by the system's kernel developers receives full trust. Many modules and drivers, however, have unknown provenance and hold only limited trust. Sentry monitors the interactions between code with low trust and critical data with high trust.

The core kernel includes operations and data exported to modules for their use as well as internal functionality and objects meant to be managed only by the core kernel itself. For example, the list of loaded modules, the process accounting list, the scheduler list, and the run queue of the Linux kernel exist in the core kernel's data and heap space and are altered by internal functionality of the kernel. However, the lack of memory barriers between the core kernel and its dynamically-loaded components prevents the kernel from disallowing illicit alterations of its internal data structures by malicious modules or drivers.

Consider as an example the Linux kernel's task management. The kernel stores per-process data, such as user IDs and group IDs that determine a process' privilege level and allowed access, in an aggregate data structure called the `task_struct`. Each `task_struct` also contains references to the next and previous task structures and thus forms a node in a doubly-linked list. Security tools in the kernel or in a monitoring virtual machine find the set of running processes by traversing the doubly-linked list [19, 35].

Suppose that a malicious application wants to execute undesirable functionality as a high-privilege user while remaining elusive from security software that searches for unexpected processes. The application may include a kernel-mode rootkit that elevates the malicious process' privilege by directly writing to its `task_struct` and hides the process by altering the kernel's process accounting list. Figure 1 shows a C code snippet from a rootkit [25] that assigns superuser privileges to its malicious process. In the code shown in Figure 2, a rootkit [40] uses the kernel macro `REMOVE_LINKS` to remove its malicious process' `task_struct` from the doubly-linked process accounting list. This macro alters the `next` and `previous` pointers within the list and allows the rootkit to hide its malicious process.

3.1 Threat Model

We assume that a powerful adversary is able to execute malicious code both at the kernel and user level, and she is able to alter critical kernel data structures at will. Kernel-level access can be achieved via kernel-level exploits or social engineering that entices a user to install a malicious kernel module, or by some other security flaws present in the system. This threat model reflects the current generation of rootkit attacks.

Due to kernel-level attacks, our defense is implanted in a hypervisor to provide tamper resistance. Virtualized environments create a security virtual machine (VM) and multiple user VMs. Our threat model includes the hypervisor and security VM as part of the trusted computing base (TCB). User VMs are untrusted and are open to both direct and indirect attacks. We do assume that the core kernel code of a user VM is protected and cannot be subverted by any malware running in the user VM. This requirement can be realized by making the kernel's code pages read-only [33]. Attacks such as Blue Pill [31] and SubVirt [21] are not possible as our threat model assumes that a trusted hypervisor is already installed, and these attacks do not handle nested virtualization [8].

3.2 Kernel Data Integrity Model

Potentially malicious kernel modules should not be able to alter security-critical data intended to be managed only by the core kernel. Sentry protects such critical data and enforces data access restrictions based upon the origin of the access within the code of the kernel and its modules or drivers. The data integrity model is straightforward and matches that of the Biba ring policy [9]:

- Trusted core kernel code may write to any kernel data.
- Loaded modules and drivers can write to any low-integrity data, which includes all driver data and portions of the core kernel data.
- Loaded modules and drivers cannot write to high-integrity data of the core kernel either through a direct write or by calling into existing code of the core kernel that will execute the write on behalf of the module, unless the control-flow transfer into the core kernel targets an exported function. Most exported functions act as *trusted upgraders*. The intent of the kernel developers was to provide APIs through which modules and drivers can legitimately make changes to critical data and leave the kernel in a consistent state.
- Exported library calls that alter raw memory, such as memcpy, memset, or bcopy, are not trusted upgraders. Since these functions can arbitrarily manipulate raw memory, changes made directly by these APIs may leave critical kernel data in an inconsistent state.

To determine if a loadable kernel component called into a core kernel function to induce alteration of high-integrity data, Sentry securely extracts the execution history from the hypervisor to identify the call chain that led to an attempted write.

Consider again the rootkit behaviors of Figures 1 and 2. When the rootkit attempts to directly modify the privileges of a process,

```
asmlinkage int give_root()
{
  if(current->uid != 0)
  {
    current->uid = 0;
    current->gid = 0;
    current->euid = 0;
    current->egid = 0;
  }
  return 0;
}
```

Figure 1: Fragment of rootkit code that elevates privileges of non-superuser processes to superuser (ID 0).

```
int init_module()
{
  task = find_task_by_pid(pid);
  if (task) {
    REMOVE_LINKS(task);
  }
}
```

Figure 2: Fragment of rootkit code that removes a malicious process identified by `pid` from the process accounting linked list.

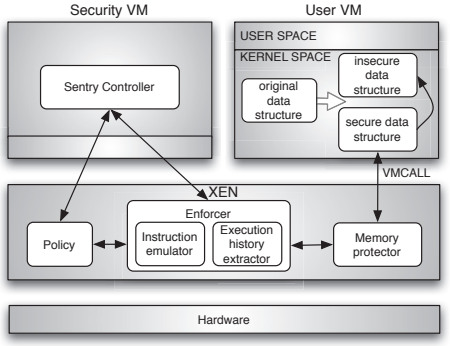


Figure 3: Sentry's architecture.

Sentry will mediate the write to security-critical data of the core kernel. The malicious code that modifies privileges by directly writing to memory is in a loaded module and not in the core kernel code, so Sentry will prevent the write (and optionally alert the system's user or administrator of a likely malware infection). Should the macro expansion of the rootkit's attempted unlinking of the `task_struct` from the linked list include function calls into internal list management functions of the kernel, then Sentry's execution history extractor will step into the code of the loaded module. Again, Sentry will deny the data write.

4. KERNEL MEMORY ACCESS CONTROL

We design a hypervisor-based kernel data protection system, *Sentry*, that protects sensitive kernel data from unauthorized modification. Its design reflects these goals:

- **Data structure protection:** Recent kernel malware instances hide their malicious activities by modifying dynamically allocated data structures using DKOM. To thwart these attacks, we develop Sentry to protect heap-allocated critical data structures.
- **Resist direct attack:** Sentry protects a kernel's dynamic data from malware. Since a rootkit executes in kernel space, any kernel-level tools can be easily subverted. Sentry's design uses a hypervisor to remain isolated from an untrusted kernel.
- **Performance:** Mediation of memory accesses may incur high performance cost if designed naïvely. To keep the overhead low, Sentry uses memory partitioning to lay out sensitive data on separate memory pages and protects those pages using the hypervisor.

Sentry's architecture consists of three components: a hypervisor, a user virtual machine (VM), and a trusted security VM (Figure 3). The hypervisor is the heart of Sentry's protections, comprised of the memory access policy description, a memory protection module, and a policy enforcement module. The automatically constructed policy statement includes a description of trusted code regions allowed to modify security-critical data. The memory protection module protects all kernel memory pages containing security-critical data. The policy enforcer mediates attempted writes to protected data and uses the policy to determine when writes should be permitted. If permitted, the enforcer's instruction emulator emulates the write operation in a manner transparent to the user VM. The enforcer also includes functionality to securely extract the execution history from the guest kernel. The user VM runs the operating system that may fall victim to kernel-level attacks. Finally, the security VM runs the controller software that starts and stops Sentry's protections and receives alerts from the hypervisor when a malicious modification is attempted.

Sentry currently supports the Xen hypervisor and Linux guest kernels, and our continuing presentation of Sentry will use Xen and Linux terminology. Sentry's techniques are general and apply to many hypervisors and operating systems.

4.1 Policy

Sentry enforces integrity protection policies based on the trust level of code attempting to alter critical data. Sentry's policies describe code regions or function call chains that are allowed to modify security-critical kernel data. Any access request that does not fall into pre-defined trusted code regions will be denied. We identify the following three types of code regions that can legitimately modify protected data:

1. All core kernel code (that is not in loadable modules or drivers) is trusted to correctly manage its own private data. This code spans memory from the Linux kernel symbol `_text` to `_etext`.
2. Kernel code from `__init_begin` to `__init_end` contains code required for the kernel to successfully boot and is likewise trusted.
3. Alterations reachable from most exported kernel functions' entry points reflect valid management of private kernel data even when a loaded component calls into the function. Exported kernel functions are deliberate APIs, defined in Linux's System.map file, created by the core developers specifically for loadable modules and drivers. Excluding library calls that alter raw memory, these functions leave an operating system's kernel in a consistent state.

4.2 Activation of Mediated Access

To enforce integrity protections, Sentry mediates all attempts to overwrite security-critical data. Sentry uses memory page access permissions to disable write permissions on any page holding sensitive kernel data. This protection provides Sentry with the ability to interpose on write accesses to protected memory pages; any write operation to a protected page causes a page fault, and on each fault the hypervisor gains the control of execution.

Sentry's memory protection relies on knowledge of the location of dynamically-allocated sensitive kernel objects. It uses code instrumentation within the user VM's kernel to activate and deactivate protections in concert with object construction and destruction. The instrumentation uses the Intel hardware virtualization instruction `VMCALL` [18], which transfers control to the hypervisor when executed. In our design, each `VMCALL` passes the page frame number (PFN) and virtual address of the newly allocated memory page requiring protection. Since the core kernel code is write-protected, as described in Section 3.1, an attacker will not be able to remove our instrumentation. On receiving a VMCALL, Sentry verifies the location of the call, and if the location does not belong to the set of locations stored in the hypervisor, then it discards the call.

Sentry receives the `VMCALL` within the hypervisor and handles the request from the guest. When the memory protection module receives a request to add protection for a guest's page, it adds the PFN to a list of protected pages, removes the pages write permission, and flushes the translation lookaside buffer (TLB) cache to remove any previous mappings that may exist for the protected PFN. When a request to remove protection on a page comes to the memory protector, it removes the PFN of the page from the list of protected pages and restores write permission. Sentry operates on the shadow page table, which is managed by the hypervisor. An attacker who is in control of guest page tables can launch memory remapping attacks. Sentry thwarts this attack by keeping both virtual and physical addresses of protected pages in the hypervisor. Since any update to guest page tables will be synced to shadow page tables, Sentry verifies whether any protected virtual addresses have been remapped. If so, Sentry protects additionally the new physical pages.

4.3 Policy Enforcement

Sentry's policy enforcer also resides in the hypervisor, and its duty is to enforce the pre-defined security policies. Both legitimate and malicious writes to protected pages will cause a page fault received by the hypervisor, providing opportunities for mediation. At each fault, the enforcer determines if it is because of Sentry's protection by verifying the PFN of the faulting page. If the PFN belongs to the list of protected PFNs, then it performs further actions. Otherwise, it directs the hypervisor to resume normal operation. On a page fault caused by Sentry's protection, the enforcer first uses the user VM's instruction pointer (`eip`) to know which code is directly attempting to write to the protected page. If the instruction pointer belongs to an untrusted code region, then the access must be denied. If the instruction pointer belongs to a trusted code region, then the enforcer must ensure that the trusted code was invoked legitimately. The enforcer securely extracts the execution history of the kernel associated with the memory write and see if the function performing the write is a trusted upgrader. If not, then untrusted code invoked an unsafe call into the kernel, and the memory alteration must be prevented. Otherwise, Sentry allows the write operation.

Once Sentry determines that a write is permitted by the integrity protection policy, it emulates the write in the same way that the guest system would have done had the protection been absent. When the emulation completes, Sentry updates the guest context registers so that the mediation of writes remains completely transparent to the guest.

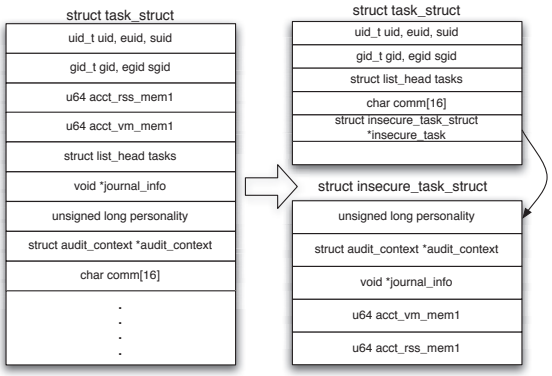


Figure 4: Memory partitioning of the Linux `task_struct` structure.

4.4 Memory Layout Optimization

The layout of kernel objects in memory challenges Sentry's ability to achieve the performance goal. Sentry's policies protect individual kernel variables and fields of aggregate structures, but it mediates writes at the granularity of complete memory pages. Structures, like `task_struct` and `module`, contain a mix of security-critical and non-critical fields laid out contiguously in memory, thereby making it difficult to provide efficient and fine-grained protection. If any variable on a page is protected, the entire page suffers the overhead of mediated writes. We would prefer that only write operations to security-sensitive members invoke a fault, thereby eliminating the performance overhead resulting from faults generated by writes to non-critical data.

To address this problem, we develop two approaches to partition a data structure, say `Obj`, into secure and insecure pieces. All security-critical members (described in Section 4.5) of `Obj` should be located together on protected pages, and all non-critical members can reside on unprotected pages.

4.4.1 Structure Division

The first approach, *structure division*, partitions `Obj` by creating a new data structure `insecure_Obj` and moving all non-sensitive members into this new structure. This procedure leaves the original `Obj` structure with security-critical members only. For example, Figure 4 shows the partial division of the Linux `task_struct` into two structures: `task_struct` (secure) and `insecure_task_struct` (insecure). A kernel using partitioned structures must allocate memory separately for the secure and insecure pieces to create different memory regions for protected and unprotected members.

To use partitioned structures, existing kernel code must be updated. All members that belong to `Obj` can still be accessed as before. However, code that accesses non-critical members must access them through `insecure_Obj`. To solve this problem, we first link `Obj` and `insecure_Obj` by adding a new pointer field in `Obj` called `insecure` that points to the `insecure_Obj` structure. Second, we updated all affected kernel code by adding an indirection through the `insecure` pointer. For example, if code in the kernel was accessing the field `journal_info` as `current->journal_info`, it is updated to become `current->insecure->journal_info`, where `current` points to the `task_struct` of the currently executing process. The process of retrofitting the existing kernel code to use a new structure can be

automated using source-to-source transformation techniques, such as those provided by CIL [24].

4.4.2 Structure Alignment

Our second partitioning approach, *structure alignment*, places security-sensitive and non-sensitive variables of a unified data structure on separate memory pages by aligning the structure in memory so that it lays across a page boundary. Only one of the two pages is protected by Sentry, and the structure's security-critical fields lay on that page. To accomplish the structure alignment, we group all security-critical members in `Obj` together at the start of the structure, group all non-critical fields at the end, and add a new alignment buffer to the structure between the two fields. This buffer is simply padding that forces all non-critical members onto the second, unprotected memory page.

These partitioning strategies have trade-offs. Structure division provides freedom in laying out the protected and unprotected structures in memory at runtime, though it needs kernel source code revision. Structure alignment has only a minimal source code change to the definition of the structure, and it could easily be integrated into an existing kernel by creating a compile-time option that inserts or removes the alignment buffer. However, it may not be an effective use of kernel memory, and it imposes constraints on runtime memory layout. We demonstrate in Section 5 that both designs are feasible, even on pervasive kernel data structures.

4.5 Identifying Security-Critical Members

Our memory layout optimization design depends upon the identification of security-critical and non-critical members in a data structure. We define a member in a data structure to be security-critical if:

- It is manipulated by attackers to carry out malicious activities. This approach provides a reasonable idea of data structure fields commonly modified by attackers and needing immediate protection. For example, many Linux-based rootkits modify uid and gid fields in a `task_struct` to elevate privileges of their malicious process. In another example, they hide the presence of their malicious modules by modifying the `next` and `prev` pointers of a `module` structure. Based on this notion, we collected rootkits to identify kernel variables that they alter.
- Subject-matter experts, such as core kernel developers, identify the variable as security critical. They can identify important members in a kernel data structure during a development phase before they are misused by attackers.
- Defensive systems rely on the variable's integrity in order to understand the security state of the user VM. For example, VMI applications [19, 35] rely on the integrity of process accounting lists and filesystem structures when constructing a trusted view of the system.

We believe that these criteria offer sufficient ability to identify security-sensitive members present in kernel data structures. Alternatively, operating system developers could identify low privilege data that is allowed to be modified by all drivers, and all other data could be protected.

5. IMPLEMENTATION

We developed Sentry using Linux 2.6 guest operating systems and the Xen hypervisor. We describe low-level implementation details of our system in this section.

5.1 Data Structure Layout

We applied our partitioning strategies to two important Linux kernel data structures: `task_struct` and `module`. We chose these two data structures due to their complexity, their relevance to current kernel-level attacks, and their pervasiveness in the kernel. The task data structure is important to the kernel because it is the fundamental unit of execution, and its complexity is based on the fact that it contains 122 members. The module data structure has 29 members, and it is used when any driver or module is loaded or unloaded, and whenever any subsystem of the kernel implemented as a driver, such as a filesystem, disk, or network device, is accessed.

To demonstrate the feasibility of our two partitioning strategies, we partitioned each of the two structure types with different techniques. We applied structure division to the widely-used `task_struct` and structure alignment to the `module` structure.

5.1.1 Partitioning of `task_struct`

To apply our partitioning strategy to the `task_struct` structure, we first identified its security-critical members. As described in Section 4.5, we identified critical members by analyzing rootkits and the Linux kernel source code. We categorized 28 of 122 members as critical and chose those for protection and partitioning. We divided `task_struct` into two parts: `task_struct` containing security sensitive members, and `insecure_task_struct` containing all non-sensitive fields.

Before partitioning, memory was allocated to an instance of the `task_struct` via `kmem_cache_alloc`. This function returns a pointer to an object of type `task_struct` from the slab cache [10]. The retrieved memory block might cross page boundaries, consequently making it difficult to provide protection for only those members that need it. Hence, we changed the memory allocation to instead use `get_free_pages` and `kmalloc`. Using `get_free_pages`, we allocated each `task_struct` on a complete page, thereby separating the critical members from non-critical fields in the kernel memory; we allocated each instance of `insecure_task_struct` using `kmalloc`. As described previously, we connected the two substructures by maintaining a reference from `task_struct` to `insecure_task_struct`. Finally, we modified Linux's `free_task` function to deallocate the memory pages allocated to `task_struct` and `insecure_task_struct` separately. After the structure partitioning, we updated the Linux kernel source code in order to work with the partitioned structure. Our source code modification to the Linux 2.6 kernel altered 2536 SLOC, which is 0.036% of the total number of lines of code in the kernel (7,041,452 SLOC).

5.1.2 Partitioning of `module`

We partitioned Linux's `module` structure using our structure alignment technique. We categorized 2 of the structure's 29 members as critical and separated them from the non-critical members with an alignment buffer that places the critical fields on a different page than non-critical fields at runtime. We first grouped all the security-critical members in the module data structure together by reorganizing the data structure. Our new alignment field provided padding that filled the rest of the memory page and caused the non-critical members to cross the page boundary to a new, unprotected page. This alternative partitioning technique did not require creation of a new insecure structure as done for the `task_struct`. The approach required no source code modification beyond the addition of the alignment buffer and required only a straightforward recompilation of the kernel due to the changed field offsets within the `module` structure.

5.2 Access Mediation and Policy Enforcement

The memory protection system of Sentry operates in two phases, both occurring concurrently: a management phase when the kernel adds or removes a page frame number (PFN) to or from the list of protected PFNs, and the mediation phase providing the actual memory protection. To perform addition, removal, and lookup operations on a PFN, we created new APIs inside the Xen hypervisor. The API includes functions `addPFNToDB`, `removePFNFromDB`, and `checkPFNInDB`, providing Sentry with the ability to add, remove, and find a protected PFN, respectively.

The second phase works by modifying the shadow page table (SPT) code of Xen. The SPT is the native page table used by the memory management hardware and managed by Xen. To provide memory protection, we modify the `__sh_propagate` function. When a new memory page is added to the guest page table, `__sh_propagate` propagates this entry to the SPT to keep both the tables in sync. While propagating this update, the memory protection system of Sentry intercepts the update and checks whether the propagation involves a protected page. If a page belongs to the list of protected pages, it removes the write permission from the page by setting the page write bit to zero in the SPT.

To intercept subsequent write faults on a protected page, we modified the shadow page fault handler function called `sh_page_fault`. When faults occur, Sentry's code inside `sh_page_fault` verifies whether or not the fault is a result of its protection mechanism. The verification dictates how the fault is to be processed by Sentry. If the fault is due to some other activity in the guest, then Sentry ignores it and resumes the guest's normal operation. Otherwise, Sentry looks into the fault to determine what code region is attempting to write to the protected page, as described in Section 4.3.

5.3 Instruction Emulation

When the kernel memory access control policy permits a mediated write operation, Sentry must reproduce the effects of the operation in a guest operating system's memory. This functionality resides in Sentry because the guest operating system cannot execute the write operation itself due to the write protection bit set on the faulted page. We implemented an instruction emulator inside Xen to perform the emulation of memory writes. With this emulator, Sentry can replay attempted writes when subsequently determined legitimate. To emulate an instruction, Sentry needs to first locate the instruction and then fetch it from guest memory. To achieve this, Sentry uses Xen's function called `hvm_copy_from_guest_virt` that reads and writes to arbitrary guest locations.

When a faulting instruction is fetched, the emulator decodes and executes the instruction inside Xen. Depending upon the instruction type, the decoding process may identify source and destination operands. The emulator executes the instruction by reading and writing the memory locations directly from the hypervisor. To ensure transparency to the guest operating system, it updates all context registers including the instruction pointer to point to the next instruction.

5.4 Secure Execution History Extraction

To determine the code that attempts to write on the protected data, we securely extracts the execution history associated with the write operation. The execution history extractor of Sentry performs a walk from the hypervisor on the kernel stack of the guest operating system by mapping the guest kernel's stack pages into the hypervisor's memory and then traversing stack frames present on the pages. Performing a walk on the kernel stack is challenging due to the unstructured layout of call stacks on x86 systems and the presence of interrupt stack frames on a kernel stack. To successfully extract stack frames, we compiled our guest kernel with a compile-time option that produces kernel code maintaining a stack frame base pointer (`ebp` register) throughout the kernel's execution. Fortunately, interrupt stack frames do not pose problems for Sentry because Sentry only performs stackwalks following a page fault that occurred due to its protection; it does not perform a stackwalk on the kernel stack at arbitrary points of the kernel's execution. Sentry additionally pauses the guest kernel while executing the stackwalk to ensure that no modifications occur while reading the guest's memory state.

When a page fault occurs, the extractor finds the location of the current stack frame from the `ebp` register. It subsequently determines the location of the return address by adding 4 bytes to the value of `ebp`. To get the actual return address, the extractor reads the value present on the computed return address location. To get the previous frame, it extracts the address stored at the location pointed to by the `ebp` register's contents. The extractor repeats this process inside the hypervisor until it reaches the end of the stackwalk.

Sentry extracts the execution history from the guest kernel stack. There are other hypervisor-based systems, such as Gateway [37] and HUKO [43], through which execution history can be extracted without relying on the kernel stack. For example: Gateway isolates drivers in an address space separate from the core kernel and monitors the kernel code invocation by drivers through the exported interface. With this design, whenever a driver enters into the core kernel through the exported interface, the transition will be monitored by the system. This information can be used by Sentry to determine the origin of request to update the critical kernel data. Another system HUKO protected guest kernel-stack memory pages using the hypervisor, allowing the secure extraction of execution history from the untampered stack. There are other hardware-based mechanisms such as Last-Branch-Record (LBR) [18, 34] and Branch tracing (BT) [1] that could be used to determine the execution history with some extra overhead. We plan to integrate these alternative designs in our future implementation.

Table 1: Sentry's attack detection results against Linux-based rootkits that modify the kernel's process and module data structures. A $\surd$ indicates that the rootkit exhibited a particular malicious behavior.

Name	*Hidden Process*	*Hidden Module*	*Privilege Escalation*	*Result*
hp	$\surd$			Detected
all-root			$\surd$	Detected
kbd-version2			$\surd$	Detected
kbd-version3			$\surd$	Detected
override			$\surd$	Detected
synapsys			$\surd$	Detected
rkit			$\surd$	Detected
lvtes		$\surd$	$\surd$	Detected
adore-ng		$\surd$	$\surd$	Detected

6. EVALUATION

In this section, we evaluate the attack detection capability, measure the performance overhead, and perform the false positive analysis of Sentry.

6.1 Attack Prevention and Detection

Sentry prevents unauthorized modification of security-critical kernel data structures. Note that Sentry is able to protect both static

and dynamic kernel data structures. However, our experiments focus on rootkit identification based upon their attempted alteration of dynamic kernel data, as this represents a significant new advancement in defensive capabilities.

We tested Sentry against a collection of DKOM rootkits present in the wild. Table 1 shows our Linux rootkit samples and the malicious behaviors that they exhibit. During our testing, we ran each rootkit sample in the user VM, which was running our Linux kernel with partitioned `task_struct` and `module` data structures. Sentry's protections started at kernel boot so that all processes beginning with the `init` process and all modules can be protected. Our results, shown in Table 1, indicate that Sentry provided a 100% detection rate for DKOM rootkits.

6.2 Performance

We carried out several experiments to measure the performance overhead incurred by the protection offered by Sentry as well as to show the effect of partitioning. Our results were measured on an Intel Core 2 Quad 2.66 GHz system with Fedora in the security VM and our partitioned 32-bit Linux kernel in the user VM. We assigned 1GB of memory to the user VM and 3GB total to the security VM and Xen. Unless otherwise specified, we performed the experiments reported in tables five times and present the median value. We show the results of some experiments using boxplots due to measurement variations common to virtualized environments. In all the experiments, "normal" refers to measurements that have no kernel memory protection, "protected" includes memory protection without partitioning, and "Sentry" includes both memory protection and partitioning.

We first evaluated the effect of protection and partitioning of module structures on Linux modules by measuring the time taken by module load (`insmod`) and unload (`rmmod`) operations. We wrote a sample module that traverses the list of loaded modules and inserted it using the `insmod` program. The same module is then unloaded using the `rmmod` program. Figures 5 and 6 present the results, and it can be seen that the loading and unloading time is higher for the unpartitioned kernel when compared with Sentry's time.

In the next set of experiments, we measured the effect of Sentry on filesystem cache performance. To test this, we used a filesystem benchmark called `IOzone`[1], which measures the throughput of read, write, re-read, and re-write operations. Figure 7 shows quartile measurements of ten repetitions of each test. Note that these results show use of the kernel's filesystem cache—rather than of disk operations—due to high variability of disk performance measurements in a virtualized environment. Sentry's protection incurs less overhead on file cache operations when performed using the partitioned kernel as compared to its effect on unpartitioned kernel, and in many cases, its performance nears that of the unprotected system.

Next, we measured the memory usage of both the partitioned and unpartitioned kernels. We performed this experiment by iterating through the guest kernel's page tables. At any given time, the partitioned kernel used 6502 pages as compared to 6302 pages used by the unpartitioned kernel. Though this extra memory is less than 1MB, it can be further reduced by using an improved memory allocator.

We next measured Sentry's performance on real world software by testing it with full applications. In our experiments, we performed compilation of a stripped down version of the Linux kernel, compilation of the Apache webserver, compression and decompression of a 225 MB file, and a network copy operation using wget. Table 2 shows the results of experiments, and it is evident that Sentry's partitioning has significantly reduced the overhead caused by its protection when compared with the overhead on an unpartitioned kernel.

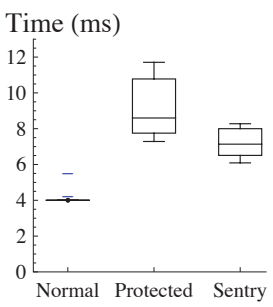


Figure 5: Module loading operation performed via `insmod`; smaller measurements are better.

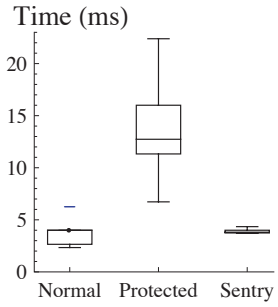


Figure 6: Module unloading operation performed via `rmmod`; smaller measurements are better.

Finally, we measured the effect of protection and partitioning of a process data structure. We measured the effect with two tests that exercise heavy legitimate use of the protected structures—process creation and context switch time—using the lmbench[2] Linux benchmark tool. Our results, shown in Table 3, indicate that Sentry incurs low overhead on a partitioned kernel as compared to the overhead on an unpartitioned kernel. Though this overhead of Sentry is more in percentage when compared with the "Normal VM" time, it gets amortized across the lifetime or execution of a process.

Although we have not explored many optimizations, the above results provide a strong evidence that Sentry can provide a balance between security and performance. Its performance on several applications is encouraging and suggests that this approach is practical.

6.3 False Positive Analysis

A false positive in our approach occurs when a security-critical member is modified by a benign module or driver that violates our integrity policies described in Section 4.1. Our analysis revealed the following:

- There were no instances when security-critical kernel data protected by Sentry was *directly* modified by a benign driver.
- Whenever security-critical data protected by Sentry was altered by a benign driver, it was done using *trusted upgraders* designed by kernel developers for that purpose, and they left the kernel in a consistent state. We illustrate this point with an example: a `task_struct` contains a member called `run_list`, which is similar to `tasks` (pointer to accounting list), but contains next and previous pointers for job scheduling; Sentry protects the `run_list` member. These pointers are modified by functions such as `enqueue_task` and `dequeue_task`, which in turn are called from the `schedule` function, which is a trusted upgrader. The `schedule` function is invoked on each context-switch and modifies the run list; it is also invoked from all modules. Since our policy allows changes made to kernel data via trusted upgraders, whenever members such as `run_list` were modified, we verified the call-chain and allowed the modification.

[1] `http://www.iozone.org`

[2] `http://www.bitmover.com/lmbench/`

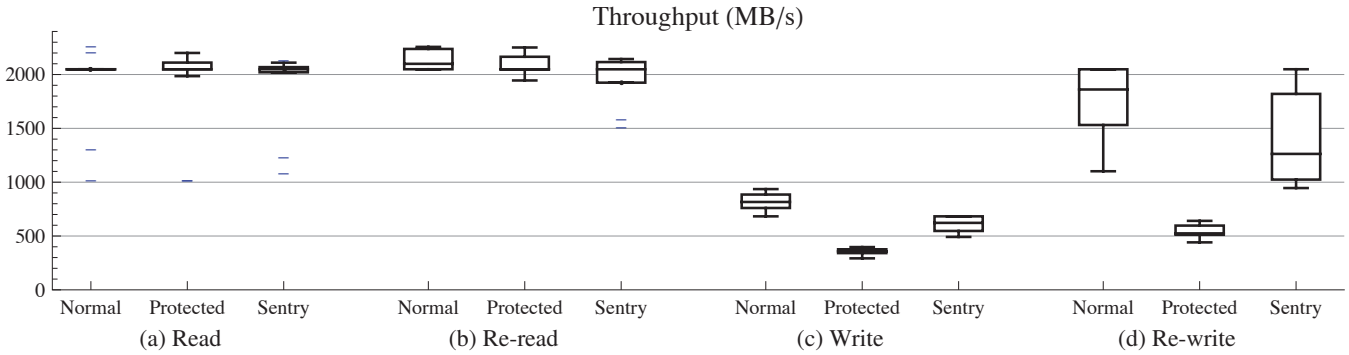


Figure 7: Performance impact of kernel memory protection on use of the kernel's file cache. All measurements show throughput in MB/s; higher measurements are better. Boxes show medians and first and third quartiles. Outliers appear as dashes. Groupings show performance of (a) file-cache cold reads, (b) cache-warm reads, (c) cache-cold writes, and (d) cache-warm writes.

Table 2: Performance impact of Sentry on real-world applications; smaller measurements are better.

Operations	*Normal VM (sec)*	*Protected (sec)*	***% Overhead***	*Sentry (sec)*	***% Overhead***
Compression	41.91	43.41	3.58	42.03	0.29
Decompression	10.09	11.68	15.76	10.11	0.20
Network File Transfer	18.46	19.45	3.85	18.52	0.33
Kernel Compilation	56.39	65.47	16.10	58.45	3.65
Apache Compilation	5.88	7.42	26.19	6.06	3.06

Table 3: Process creation and context-switch time measured with lmbench; smaller measurements are better. The default CPU time-slice on the test system was 100 ms.

Operations	*Normal VM (μs)*	*Protected (μs)*	***Overhead (%)***	*Sentry (μs)*	***Overhead (%)***
Process fork+exit	342.89	1255.89	266.27	550.52	60.55
Context switch	1.85	41.14	2123.78	6.45	248.65

With the above design in place, our system did not show any false positives and detected all attacks.

6.4 Discussion

Using Sentry, we demonstrated how to protect kernel data structures. The current prototype of Sentry only protects two key structures. We chose these data structures because of their size, pervasiveness, and importance. However, there are other kernel data structures that may also require similar protection. Our system can be extended to protect other data structures. Our performance results show that Sentry imposes very low overhead on the two protected structures, and we expect that Sentry will maintain its low overhead even when it extends its protection to other structures. However, the actual performance overhead depends upon the number of sensitive members, and the rate at which the kernel and its components legitimately modify these sensitive members in newly-protected structures.

7. CONCLUSIONS

We developed Sentry to provide partitioned kernel memory in a manner similar to memory isolation provided by the kernel for its applications. We protect security-critical data by protecting memory pages containing that data. To provide balance between security and performance, we altered the kernel memory layout to aggregate data needing the same policy enforcement on the same memory pages. Sentry's security evaluation shows that the system is capable of detecting attacks against dynamic kernel data, and its performance evaluation shows low overheads on microbenchmarks and real-world applications.

Acknowledgement

We thank the anonymous reviewers for their comments on the paper. We also thank Ikpeme Erete for his assistance with this project. This work was partly supported by National Science Foundation contract number CNS-0845309. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of their employers, the NSF, or the US Government.

8. REFERENCES

[1] Branch Tracing with Intel MSR Registers. `http://www.openrce.org/blog/view/535/Branch_Tracing_with_Intel_MSR_Registers`. Last Accessed Sep. 15, 2012.

[2] New for Kernel-Mode Driver Architecture. `http://msdn.microsoft.com/en-us/library/windows/hardware/hh439748%28v=vs.85%29.aspx`. Last Accessed Sep. 15, 2012.

[3] Windows 8 Security: What's New. `http://www.pcworld.com/article/255776/windows_8_security_whats_new.html`. Last Accessed Sep. 15, 2012.

[4] Windows ISV Software Security Defenses. `http://msdn.microsoft.com/en-us/library/bb430720`. Last Accessed Sep. 15, 2012.

[5] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ASPLOS*, San Jose, CA, Oct. 2006.

[6] A. Baliga, V. Ganapathy, and L. Iftode. Automatic inference and enforcement of kernel data structures invariants. In

ACSAC, Anaheim, CA, Dec. 2008.

[7] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *ACM SOSP*, Bolton Landing, NY, Oct. 2003.

[8] M. Ben-Yahuda, M. D. Day, Z. Dubitsky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B. Yassour. The Turtles project: Design and implementation of nested virtualization. In *OSDI*, 2010.

[9] K. J. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, Mitre, Apr. 1977.

[10] J. Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX*, Boston, MA, June 1994.

[11] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast byte-granularity software fault isolation. In *ACM SOSP*, Big Sky, Montana, Oct. 2009.

[12] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *ACM SOSP*, Stevenson, WA, Oct. 2007.

[13] F. Corbato and V. Vyssotsky. Introduction and overview of the Multics system. In *Fall Joint Computer Conference*, Las Vegas, NV, Nov. 1965.

[14] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: Malware analysis via hardware virtualization extensions. In *ACM CCS*, Alexandria, VA, Oct. 2008.

[15] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *OSDI*, Seattle, WA, Nov. 2006.

[16] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *ACM SOSP*, Bolton Landing, NY, Oct. 2003.

[17] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *NDSS*, San Diego, CA, Feb. 2003.

[18] Intel. *System Programming Guide: Part 2*. Intel 64 and IA-32 Architectures Software Developer's Manual, 2004.

[19] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through VMM-based 'out-of-the-box' semantic view. In *ACM CCS*, Alexandria, VA, Nov. 2007.

[20] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. VMM-based hidden process detection and identification using Lycosid. In *ACM VEE*, Seattle, WA, Mar. 2008.

[21] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. SubVirt: Implementing malware with virtual machines. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2006.

[22] L. Litty, H. A. Lagar-Cavilla, and D. Lie. Hypervisor support for identifying covertly executing binaries. In *USENIX Security Symposium*, San Jose, CA, Aug. 2008.

[23] Microsoft. PatchGuard. `http://blogs.msdn.com/windowsvistasecurity/archive/2006/08/11/695993.aspx`. Last accessed Sep. 15, 2012.

[24] G. C. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Conference on Compiler Construction (CC)*, Grenoble, France, Apr. 2002.

[25] Packet Storm. All-root. `http://packetstormsecurity.org/UNIX/penetration/rootkits/all-root.c`. Last accessed Sep. 15, 2012.

[26] B. D. Payne, M. Carbone, and W. Lee. Secure and flexible monitoring of virtual machines. In *ACSAC*, Miami, FL, Dec. 2007.

[27] B. D. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2008.

[28] N. L. Petroni, Jr., T. Fraser, A. Walters, and W. A. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *USENIX Security Symposium*, Vancouver, BC, Canada, Aug. 2006.

[29] N. L. Petroni, Jr. and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *ACM CCS*, Alexandria, VA, Nov. 2007.

[30] R. Riley, X. Jiang, and D. Xu. Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing. In *RAID*, Boston, MA, Sept. 2008.

[31] J. Rutkowska. Subverting Vista kernel for fun and profit. In *Black Hat USA*, 2006.

[32] R. Sailer, T. Jaeger, E. Valdez, R. Caceres, R. Perez, S. Berger, J. L. Griffin, and L. van Doorn. Building a MAC-based security architecture for the Xen open-source hypervisor. In *ACSAC*, Tucson, AZ, Dec. 2005.

[33] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *ACM SOSP*, Stevenson, WA, Oct. 2007.

[34] M. L. Soffa, K. R. Walcott, and J. Mars. Exploiting hardware advances for software testing and debugging. In *ICSE*, Honolulu, HI, May 2011.

[35] A. Srivastava and J. Giffin. Tamper-resistant, application-aware blocking of malicious network connections. In *RAID*, Boston, MA, Sept. 2008.

[36] A. Srivastava and J. Giffin. Automatic discovery of parasitic malware. In *RAID*, Ottawa, Canada, Sept. 2010.

[37] A. Srivastava and J. Giffin. Efficient monitoring of untrusted kernel-mode execution. In *NDSS*, San Diego, California, Feb. 2011.

[38] A. Srivastava, A. Lanzi, J. Giffin, and D. Balzarotti. Operating system interface obfuscation and the revealing of hidden operations. In *DIMVA*, Netherlands, July 2011.

[39] Symantec. Windows rootkit overview. `http://www.symantec.com/avcenter/reference/windows.rootkit.overview.pdf`. Last accessed Sep. 15, 2012.

[40] ubra. Process hiding and the Linux scheduler. *Phrack*, 11(63), Jan. 2005.

[41] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering kernel rootkits with lightweight hook protection. In *ACM CCS*, Chicago, IL, Nov. 2009.

[42] E. Witchel, J. Cates, and K. Asanovic. Mondrian memory protection. In *ASPLOS*, San Jose, CA, Oct. 2002.

[43] X. Xiong, D. Tian, and P. Liu. Practical protection of kernel integrity for commodity os from untrusted extensions. In *NDSS*, San Diego, California, Feb. 2011.

TrueErase: Per-File Secure Deletion for the Storage Data Path

Sarah Diesburg, Christopher Meyers, Mark Stanovich, Michael Mitchell, Justin Marshall, Julia Gould, and An-I Andy Wang
Florida State University
{diesburg, meyers, stanovic, mitchell, jmarshal, gould, awang}@cs.fsu.edu

Geoff Kuenning
Harvey Mudd College
geoff@cs.hmc.edu

ABSTRACT

The ability to securely delete sensitive data from electronic storage is becoming important. However, current per-file deletion solutions tend to be limited to a segment of the operating system's storage data path or specific to particular file systems or storage media.

This paper introduces TrueErase, a holistic secure-deletion framework. Through its design, implementation, verification, and evaluation, TrueErase shows that it is possible to build a legacy-compatible full-storage-data-path framework that performs per-file secure deletion and works with common file systems and solid-state storage, while handling common system failures. In addition, this framework can serve as a building block for encryption- and tainting-based secure-deletion systems.

Categories and Subject Descriptors

D.4.2 [**Storage Management**]: Allocation/deallocation strategies, D.4.3 [**File Systems Management**]: Access methods, D.4.6 [**Security and Protection**]: Information flow controls

General Terms

Design, Security

Keywords

Secure deletion, file systems, storage, security, NAND flash

1. INTRODUCTION

Data privacy is an increasing concern as more sensitive information is being stored in electronic devices. Once sensitive data is no longer needed, users may wish to permanently remove this data from electronic storage. However, typical file deletion mechanisms only update the file's metadata (e.g., pointers to the data), leaving the file data intact. Even recreating the file system from scratch does not ensure the data is removed [7].

A remedy is secure deletion, which should render data irrecoverable. Much secure deletion is implemented through partition- or device-wide encryption or overwriting techniques (see §2). However, coarse-grained methods may not work on media such as NAND flash [41], and are cumbersome to use when only a few files need to be securely deleted. Further, securely deleting an entire device or partition may be infeasible for embedded devices.

Per-file secure deletion is concerned with securely removing a specific file's content and metadata (e.g., name). This ability can assist with implementing privacy policies concerning the selective destruction of data after it has expired (e.g., client files), complying with government regulations to dispose of sensitive data (e.g., HIPPA [10]), deleting temporarily shared trade secrets, military applications demanding immediate destruction of selected data, and disposing of media in one-time-use applications.

Unfortunately, existing per-file secure-deletion solutions tend to be file-system- and storage-medium-specific, or limited to one segment of the operating-system storage data path (e.g., the file system) without taking into account other components (e.g., storage media type). For example, a secure deletion issued by a program might not be honored by optimization software used on typical flash devices that keep old versions of the data [41]. Solutions that rely on secure deletion of a stored encryption key become a subset of this problem, because they, too, must have a way to ensure the key is erased.

In addition, achieving secure deletion is hard due to diverse threat models. This paper focuses on dead forensics attacks on local storage, which occur after the computer has been shut down properly. Attacks on backups or live systems, cold-boot attacks [9], covert channels, and policy violations are beyond our scope.

In particular, our system assumes that we have full control of the entire storage data path in a non-distributed environment. Thus, the research question is, under benign user control and system environments, what holistic solution can we design and build to ensure that the secure deletion of a file is honored throughout the legacy storage data path? Although tightly constrained, this criterion still presents significant challenges.

We introduce TrueErase, a framework that irrevocably deletes data and metadata upon user request. TrueErase goes to the heart of the user's mental model: securely deleted data, like a shredded document, should be irrecoverable.

We assume the presence of file-system-consistency properties [34], which have been shown to be a requirement of secure deletion. We also note that if we have control over the storage layer, achieving raw NAND flash secure deletion is straightforward [18, 27, 38, 41].

However, TrueErase is designed to overcome the many challenges of correctly propagating secure deletion information in a full-data-path manner—all the way from the user to the storage. This framework is essential for building other secure-deletion capabilities that rely on tainting or encryption-based key deletion. Thus, our contributions are the following: (1) a per-file secure-deletion framework that works with the legacy storage data path, which (2) addresses the challenges raised throughout the data path into a single work and (3) has been systematically verified.

2. EXISTING APPROACHES

We distinguish the need for TrueErase from existing approaches by enumerating desirable characteristics of secure deletion systems (Table 1).

- ***Per-file:*** Fine-granularity secure deletion brings greater control to the user while potentially reducing costly secure deletion operations on the storage device.

ACSAC '12, Dec. 3–7, 2012, Orlando, Florida, USA.

- ***Encryption-free:*** Encryption systems securely delete by destroying the keys of encrypted files. However, these systems are complex due to managing per-file keys, providing efficient random access, data and metadata padding and alignments, etc. Encryption may also expire due to more computing power (e.g., quantum computers) or implementation bugs [4, 5]; thus, encrypted data might still be recovered after its key is deleted. In addition, passphrase-derived keys can be surrendered (e.g., via coercion) to retrieve deleted files. For these reasons, systems that keep keys on storage or third-party sites still need a way to securely delete keys on various storage media.
- ***Data-path-wide:*** Many secure-deletion solutions are within one layer of the legacy storage data path (Figure 1). The storage-management layer has no information about a block's file ownership [6, 35] and cannot support per-file deletion. File-system- and application-layer solutions are generally unaware of the storage medium and have limited control over the storage location of data and metadata. In addition, storage such as NAND flash may leave multiple versions of sensitive data behind. Thus, we need a solution that spans the entire data path to enforce secure deletion.
- ***Storage-medium-agnostic:*** A secure-deletion framework should be general enough to accommodate solid-state storage and portable devices that use these types of storage.
- ***Limited changes to the legacy code:*** Per-file secure deletion requires some information from the file system. However, getting such information should not involve modifying thousands of lines of legacy code.
- ***Metadata secure deletion***: Secure deletion should also remove metadata, such as the file name, size, etc.
- ***Crash handling:*** A secure-deletion solution should anticipate system crashes and provide meaningful semantics afterwards.

Table 1: Existing secure-deletion methods and characteristics. columns: F. per-file; E. encryption-free; D. data-path-wide; S. storage-medium-agnostic; L. limited changes to legacy code; M. securely delete metadata; C. handle crashes

	F	E	D	S	L	M	C
Secure delete encrypted device/partition key [12, 13, 19, 30, 40]					✓	✓	✓
Specialized hard drive commands [11]		✓			✓	✓	✓
Specialized flash medium commands (page granularity) [41]	✓	✓			✓		
Stackable file system deletion [14, 15]	✓	✓			✓	✓	✓
Modified file system – deletion through overwriting [1, 14, 15]	✓	✓			✓	✓	✓
Modified file system – deletion through encryption [25]	✓				?		?
Dedicated server(s) for encryption keys [8, 24]	✓				✓	?	✓
Encrypted backup system [2]	✓				✓		?
User-space solution on top of flash file system [27]	✓	✓			✓	?	
Overwriting tools [19, 28, 29, 31, 42]	✓	✓			✓		
Modified flash file systems – device erasures and/or overwriting [27, 38]	✓	✓	✓		✓	?	?
Modified flash file systems – encryption with key erasure [18]	✓		✓		?	✓	?
Semantically-Smart Disk Systems [35, 36]	✓	✓	✓		✓		✓
Type-Safe Disks [33]	✓	✓	✓				✓
Data Node Encrypted File System [26]	✓	✓	✓		✓	✓	✓
TrueErase	✓	✓	✓	✓	✓	✓	✓

3. CHALLENGES AND ASSUMPTIONS

Designing and implementing per-file secure deletion is challenging for a number of reasons:

- ***No pre-existing deletion operation:*** Other than removing references to data blocks and setting the file size and allocation bits to zeros, file systems typically do not issue requests to erase file content. (Note that the ATA Trim command was not implemented for security [32] and might not delete all data [17].)
- ***Complex storage-data-path optimizations***: Secure deletion needs to retrofit legacy asynchronous optimizations. In particular, storage requests may be reordered, concatenated, split, consolidated (applying one update instead of many to the same location), cancelled, or buffered while in transit.
- ***Lack of data-path-wide identification***: Tracking sensitive data throughout the operating system is complicated by the possible reuse of data structures, ID numbers, and memory addresses.
- ***Verification***: Although verification is often overlooked by various solutions, we need to ensure that (1) secure deletions are correctly propagated throughout the storage data path, and (2) assumptions are checked when possible.

Due to these challenges, we assume a user has administrative control of an uncompromised, single-user, single-file-system, non-RAID, non-distributed system. The threat model is dead forensics attacks, which occur after a user unmounts and shuts down the system after completing secure-deletion operations.

Our system assumes control of the entire storage data path. Although assuming access to proprietary firmware may be optimistic, this paper argues that there is a need for storage devices to expose information and control to support secure-deletion features correctly.

In addition, we assume that common journaling file systems that adhere to the consistency properties specified by [34] are used, since we cannot verify secure deletion in file systems that cannot even guarantee their own consistency (e.g., ext2, FAT). Further, all update events and block types are reported to our framework to verify that we are tracking all important events. These assumptions allow us to focus on building and verifying the properties of secure deletion.

4. TRUEERASE DESIGN

We introduce TrueErase (Figure 1), a holistic framework that propagates file-level information to the block-level storage-management layer via an auxiliary communication path, so that per-file secure deletion can be honored throughout the data path.

Our system is designed with the following observations and formulated guidelines:

- Modifying the request ordering of the legacy data path is undesirable because it is difficult to verify the legacy semantics. Thus, we leave the legacy data flow intact.
- Per-file secure deletion is a performance optimization over applying secure deletion to all file removals. Thus, we can simplify tricky decisions about whether a case should be handled securely by handling it securely by default.
- Persistent states complicate system design with mechanisms to survive reboots and crashes. Thus, our solution minimizes the use of persistent states.

The major areas of TrueErase's design include (1) a user model to specify which files or directories are to be securely deleted, (2) tracking and propagating information across storage layers via a centralized module named ***TAP***, (3) enforcing secure deletion via storage-specific mechanisms added to the storage-management layer, and (4) exploiting file-system consistency properties to enumerate cases for verification.

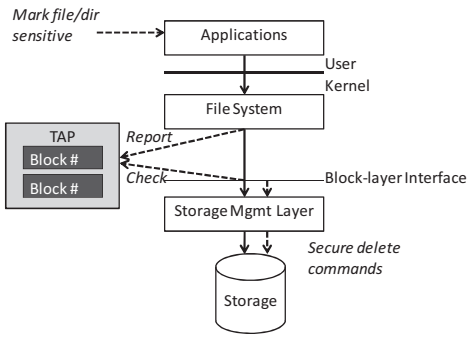


Figure 1. TrueErase framework. The shaded box and dotted lines represent our augmented data path. Other boxes and lines represent the legacy storage data path.

4.1 User Model

A naïve user may mark the entire storage device sensitive for secure deletion. A more sophisticated user can mark all user-modifiable folders sensitive. An expert user can follow traditional permission semantics and apply common attribute-setting tools to mark a file or directory as ***sensitive***, which means the sensitive file, or all files under the sensitive directory, will be securely deleted when removed. A legacy application can then issue normal deletion operations, which are carried out securely, to remove sensitive file and directory data content, metadata, and associated copies within the storage data path. However, there are deviations from the traditional permission models.

Toggling the sensitive status: Before the status of a file or directory is toggled from non-sensitive to sensitive, older versions of the data and metadata may have already been left behind. Without tracking all file versions or removing all old versions for all files, TrueErase can enforce secure deletion only for files or directories that have remained sensitive since their creation. Should a non-sensitive file be marked sensitive, secure deletion will be carried out only for the versions of the metadata and file content created after that point.

Name handling: A directory is traditionally represented as a special file, with its data content storing the file names the directory holds. Although file permissions are applied to its data content, permission to handle the file name is controlled by its parent directory.

Under TrueErase, marking a file sensitive will also cause its name stored in the parent directory to be securely handled, even if the parent is not marked sensitive. Otherwise, marking a file sensitive would require its parent to be marked sensitive for containing the sensitive file name, its grandparent to be marked sensitive for containing the sensitive parent name, etc., until reaching the root directory.

Links: Similar to legacy semantics, a secure deletion of a hard link is performed when the link count decreases to zero. Symbolic link names and associated metadata are not supported, but file data written to a symbolic link will be treated sensitively if the link's target file is sensitive.

4.2 Information Tracking and Propagation

Tracking and propagating information from the file system to the lower layers is done through a centralized type/attribute propagation (***TAP***) module.

4.2.1 Data Structures and Globally Unique IDs

TAP expands the interface between the file system and the block layer in a backward-compatible way. This passive forwarding module receives pending sensitive update and deletion events from file systems, and uses internal ***write entries*** and ***deletion reminders,*** respectively, to track these events.

Since various data structures (e.g., block I/O structures), namespaces (e.g., i-node numbers), and memory addresses can be reused, correct pairing of write entries and deletion reminders requires a unique ID scheme. In particular, the IDs need to be (1) accessible throughout the data path and (2) altered when its content association changes. TrueErase embeds a monotonically increasing globally unique page ID in each memory page structure accessible throughout the data path. The ID is altered at allocation time, so that the same page reallocated to hold versions of the same logical block has different IDs.

To handle various tracking units, such as logical blocks, requests, physical sectors, and device-specific units, TAP tracks by the physical sector, because that is unique to the storage device and can be computed from anywhere in the data path. The globally unique page ID and the physical sector number form a globally unique ID (***GUID***).

In addition to the GUID, a write entry contains the update type (e.g., i-node) and the security status. A deletion reminder contains a ***deletion list*** of physical sector numbers to be deleted.

Table 2: The TAP Interface.

TE_report_write(*GUID, block type, secure status*): Creates or updates a TAP write entry associated with the GUID.
TE_report_delete(*metadata GUID, metadata block type, metadata secure status, deletion sector number*): Creates or updates a TAP deletion reminder that contains the sector number. The reminder is attached to the write entry (created as needed) associated with the metadata GUID.
TE_report_copy(*source GUID, destination GUID, flag*): Copies information from a write entry corresponding to the source GUID to a new entry corresponding to the destination GUID. The flag determines whether deletion reminders should be transferred.
TE_cleanup_write(*GUID*): Removes the write entry with a specified GUID. This call also handles the case in which a file has already been created, written, and deleted before being flushed to storage.
TE_check_info(*GUID*): Used to query TAP to retrieve information about a block layer write with a specific GUID.

4.2.2 TAP Interface and Event Reporting

Through the TAP interface (Table 2), the file system must report all important events: file deletions and truncations, file updates, and certain journaling activities.

File deletions and truncations: Data deletion depends on the update of certain metadata, such as free-block bitmaps. TAP allows file systems to associate deletion events with metadata update events via TE_report_delete(). Within TAP, write entries are associated with respective deletion reminders and their deletion lists.

Eventually, the file system flushes the metadata update. Once the block-layer interface receives a sector to write, the interface uses the GUID to look up information in TAP through TE_check_info(). If the write entry is linked to a deletion reminder, the storage-management layer must securely delete those sectors on the deletion list before writing the metadata update.

Additionally, the storage-management layer can choose a secure-deletion method that matches the underlying medium. For NAND flash, triggering the erase command once may be sufficient (details in §4.3). For a hard drive, the sectors can be directly overwritten with random data.

After secure deletion has been performed and the metadata has been written to storage, the block-layer interface can call TE_cleanup_write() to remove the associated TAP write entries and deletion reminders.

For data-like metadata blocks (e.g., directory content), deletion handling is the same as that of data blocks.

File updates: Performing deletion operations is not enough to ensure all sensitive information is securely deleted. By the time a secure-deletion operation is issued for a file, several versions of its blocks might have been created and stored (e.g., due to flash optimizations), and the current metadata might not reference old versions. One approach is to track all versions, so that they can be deleted at secure-deletion time. However, tracking these versions requires persistent states, and thus recovery mechanisms to allow those states to survive failures.

TrueErase avoids persistent states by tracking and deleting old versions along the way. That is, secure deletion is applied for each update that intends to overwrite a sensitive block in place (***secure write*** for short). Therefore, in addition to deletion operations, TrueErase needs to track all in-transit updates of sensitive blocks.

The file system can report pending writes to TAP through TE_report_write(). Eventually, the block-layer interface will receive a sector to write and can then look up information via TE_check_info(). If the corresponding write entry states that the sector is secure, the storage-management layer will write it securely. Otherwise, it will be written normally. After the sector is written, the block-layer interface calls TE_cleanup_write().

4.2.3 Journaling and Other Events

Sometimes file blocks are copied to other memory locations for performance and accessibility reasons (e.g., file-system journal, bounce buffers, etc.). When that happens, any write entries associated with the original memory location must be copied and associated with the new location. If there are associated deletion reminders, whether they are transferred to the new copy is file-system-specific. For example, in ext3, the deletion reminders are transferred to the memory copy in the case of bounce buffers or to the memory copy that will be written first in the case of journaling. Memory copies are reported to TAP through TE_report_copy().

4.3 Enhanced Storage-management Layer

TrueErase does not choose a secure-deletion mechanism until a storage request has reached the storage-management layer. By doing so, TrueErase ensures that the chosen mechanism matches the characteristics of the underlying storage medium. For example, the process of securely erasing flash memory (erase) is quite different from the process of securely erasing information on a disk (overwrite). Users should be unaware of this difference.

In addition, we can further add different secure-deletion methods at the storage layer in accordance with different requirements and government regulations [21, 37].

TrueErase can work with both flash storage and traditional hard drives. Due to the difficulty of secure deletion on flash, we concentrate on applying TrueErase to flash storage in this paper. However, we also provide a high-level design of a hard drive solution to show the generalizability of TrueErase.

4.3.1 NAND Flash Storage Management

NAND flash basics: NAND flash has the following characteristics: (1) writing is slower than reading, and erasure can be more than an order of magnitude slower [3]; (2) NAND reads and writes are in ***flash pages*** (of 2-8 Kbytes), but erasures are performed in ***flash blocks*** (typically 64-512 Kbytes consisting of contiguous pages); (3) in-place updates are generally not allowed—once a page is written, the flash block containing this page must be erased before this page can be written again, and other in-use pages in the same flash block need to be copied elsewhere; and (4) each storage location can be erased only 10K-1M times [3].

As a common optimization, when flash receives a request to overwrite a flash page, a ***flash translation layer (FTL)*** remaps the write to a pre-erased flash page, stamps it with a version number, and marks the old page as invalid, to be cleaned later. (Flash overwrites might be allowed for some special cases.) These invalid pages are not accessible to components above the block layer; however, they can be recovered by forensic techniques [20]. To prolong the lifespan of the flash, ***wear-leveling*** techniques are often used to spread the number of erasures evenly across all storage locations.

NAND secure commands: We added two secure-deletion commands to the storage-management layer (i.e., FTL) for flash.

Secure_delete(*page numbers*): This call specifies pages on flash to be deleted securely. The call copies other in-use pages from the current flash block to other areas, and marks those pages as unused in the block. The specified pages then can be marked invalid, and the current flash block can be cleared via a flash erase command.

Secure_write(*page numbers, data*): Generally, writing to the same logical page to flash would result in a new physical page being allocated and written, with the physical page holding old data versions marked invalid. This call, on the other hand, would securely delete those invalid pages with Secure_delete().

We choose this type of secure-deleting flash behavior instead of zero-overwriting or scrubbing [27, 38, 41] for ease of implementation and portability. Alternative flash secure deletion schemes may have better performance on specific chips.

NAND garbage collection: When a NAND flash device runs low on space, it triggers wear leveling to compact in-use pages into fewer flash blocks. However, this internal storage reorganization does not consult with higher layers and has no knowledge of file boundaries, sensitive status, etc. Thus, in addition to storing a file's sensitive status in the extended attributes, we found it necessary to store a sensitive-status bit in the per-page ***control area***. (This area also contains a page's in-use status.) With this bit, when a sensitive page is migrated to a different block, the old block is erased via Secure_delete(). Consistency between the file system and the storage status is addressed in § 4.5.

4.3.2 Hard Drive Storage Management

Hard drive basics: On a hard drive, writes can be performed in-place, in that a write to a logical sector will directly overwrite the same physical sector.

Hard drive secure commands: Similarly to the NAND secure commands, we can add **secure_delete**(*sector numbers*) and **secure_write**(*sector numbers, data*) commands for disk. The results of these commands are the same as for the flash case—only the mechanism changes. The **secure_delete** command will overwrite *sector numbers* in-place with random data, and the **secure_write** command will first overwrite sectors specified by *sector numbers* with random data before writing *data*. The number of overwrites of random data is configurable. These commands can be placed in a high-level software driver, such as the Linux device-mapper layer.

4.4 File-system-consistency Properties

We cannot guarantee the correctness of our framework if it is required to interact with file systems that exhibit arbitrary behaviors. Therefore, we applied the consistency properties of file systems defined in [34] to enumerate corner cases. (These

properties also rule out ext2 and FAT, which are prone to inconsistencies due to crashes.) By working with file systems that adhere to these properties, we can simplify corner-case handling and verify our framework systematically.

In a simplified sense, as long as pieces of file metadata reference the correct data and metadata versions throughout the data path, the system is considered consistent. In particular, we are interested in three properties in [34]. The first two are for non-journaling-based file systems. In a file system that does not maintain both properties, a non-sensitive file may end up with data blocks from a sensitive file after a crash recovery. The last property is needed only for journaling-based file systems.

- ***The reuse-ordering property*** ensures that once a file's block is freed, the block will not be reused by another file before its free status becomes persistent. Otherwise, a crash may lead to a file's metadata pointing to the wrong file's content. Thus, before the free status of a block becomes persistent, the block will not be reused by another file or changed into a different block type. With this property, we do not need to worry about the possibility of dynamic file ownership and types for in-transit blocks.
- ***The pointer-ordering property*** ensures that a referenced data block in memory will become persistent before the metadata block in memory that references the data block. With this ordering reversed, a system crash could cause the persistent metadata block to point to a persistent data block location not yet written. This property does not specify the fate of updated data blocks in memory once references to the blocks are removed. However, the legacy memory page cache prohibits unreferenced data blocks from being written. The pointer-ordering property further indicates that right after a newly allocated sensitive data block becomes persistent, a crash at this point may result in the block being unreferenced by its file. To cover this case, we will perform secure deletion on unreferenced sensitive blocks at recovery time (see §4.5).
- ***The non-rollback property*** ensures that older data or metadata versions will not overwrite newer versions persistently—critical for journaling file systems with versions of requests in transit. That is, we do not need to worry that an update to a block and its subsequent secure deletion will be written persistently in the wrong order.

With these consistency properties, we can identify the structure of secure-deletion cases and handle them by: (1) ensuring that a secure deletion occurs before a block is persistently declared free, (2) the dual case of hunting down the persistent sensitive blocks left behind after a crash but before they are persistently referenced by file-system metadata, (3) making sure that secure deletion is not applied (in a sense, too late) to the wrong file, (4) the dual case of making sure that a secure block deletion is not performed too early and gets overwritten by a buffer update from a deleted file, and (5) handling in-transit versions of a storage request (mode changing, reordering, consolidation, merging, and splitting). Buffering, asynchrony, and cancelling of requests are handled by TAP.

4.5 Other Design Points

Crash handling: TAP contains no persistent states and requires no additional recovery mechanisms. Persistent states stored as file-system attributes are protected by journal-recovery mechanisms. At recovery time, a journal is replayed with *all* operations handled securely. We then securely delete the entire journal. To hunt down leftover sensitive data blocks, we sequentially delete sensitive blocks that are not marked as allocated by the file system for flash and disk [1]. Since flash migrates in-use pages from to-be-erased blocks by copying those pages elsewhere before erasing the old versions, some sensitive pages may have duplicates during a crash. Given that the secure deletion of the page did not complete, the common journal crash-recovery mechanism will reissue the operation, so that the other remaining in-use pages in the same flash block can continue the migration, and the block can then be erased.

***Consolidation of requests*:** When consolidation is not permitted (e.g., consolidations of overwrites on storage), we need to disable storage-built-in write caches or use barriers and device-specific flush calls to ensure that persistent updates are achieved [22]. When consolidation is permitted (such as in the page cache or journal), we interpret an update's sensitive status during consolidation conservatively. As long as one of the updates to a given location is sensitive, the resulting update will be sensitive.

Dynamic sensitive-mode changes for in-transit blocks: To simplify tracking the handling of a block's sensitive status, we allow a non-sensitive in-transit file or directory to be marked sensitive, but a sensitive object is not allowed to be marked non-sensitive.

Shared block security status: A metadata block often stores metadata for many files, probably with mixed sensitive status. Thus, updating non-sensitive metadata may also cause the sensitive metadata stored on the same block to appear in the data path. We simplify the handling of this case by treating a shared metadata block sensitively as long as one of its metadata entries is sensitive.

Partial secure deletion of a metadata sector: Securely deleting a file's metadata only at the storage level is insufficient, since a metadata block shared by many files may still linger in the memory containing the sensitive metadata. If the block is written again, the metadata we thought securely deleted will return to storage. In these cases, we zero out the sensitive metadata within the block in memory, and then update the metadata as a sensitive write.

5. TRUEERASE IMPLEMENTATION

We prototyped TrueErase under Linux 2.6.25.6 and applied our framework to the popular ext3 and jbd journaling layer due to their adherence to file-system-consistency properties [34]. Because raw flash devices and their development environments were not widely available when we began our research, we used SanDisk's DiskOnChip flash and the associated Inverse NAND File Translation Layer (INFTL) kernel module as our FTL. Although DiskOnChip is dated, our design is applicable to modern flash and development environments. As future work, we will explore newer environments, such as OpenSSD [39].

Overall, our user model required 198 lines of C code; TAP, 939; secure-deletion commands for flash, 592; a user-level development environment for kernel code, 1,831; and a verification framework, 8,578.

5.1 Secure-deletion Attributes

A user can use the legacy `chattr +s` command to mark a file or directory as sensitive. However, by the time a user can set attributes on a file, its name may already be stored non-sensitively. Without modifying the OS, one remedy is to cause files or directories under a sensitive directory to inherit the attribute when they are created. We also provide `smkdir` and `screat` wrapper scripts that create a file or directory with a temporary name, mark it sensitive, and rename it to the sensitive name.

5.2 TAP Module

TAP is implemented as a kernel module. We will give a brief background on ext3 and jbd to clarify their interactions with TAP.

5.2.1 *Background on Ext3, Journaling, and Jbd*

File truncation/deletion under ext3: Ext3 deletes the data content of a file via its truncate function, which involves updating ***(t1)*** the i-node to set the file size to zero, ***(t2)*** metadata blocks to remove pointers to data blocks, and ***(t3)*** the bitmap allocation blocks to free up blocks for reuse. Multiple rounds of truncates may be required to delete the content of a large file.

Deleting a file involves: removing the name and i-node reference from the directory, adding the removed i-node to an orphan list; truncating the entire file via (t1) to (t3); removing the i-node from the orphan list; and updating the i-node map to free the i-node.

Journaling: Typical journaling employs the notion of a transaction, so the effect of an entire group of updates is either all or nothing. With group-commit semantics, the exact ordering of updates within a transaction (an update to an i-node allocation bitmap block) may be relaxed while preserving correctness, even in the face of crashes. To achieve this effect, all writes within a transaction are ***(j1)*** journaled or committed to storage persistently; then ***(j2)*** propagated to their final storage destinations, after which ***(j3)*** they can be discarded from the journal.

A committed transaction is considered permanent even before its propagation. Thus, once a block is committed to be free (j1), it can be used by another file. At recovery time, committed transactions in the journal are replayed to re-propagate or continue propagating the changes to their final destinations. Uncommitted transactions are aborted.

Jbd: Jbd differentiates data and metadata. We chose the popular ordered mode, which journals only metadata but requires ***(j0)*** data blocks to be propagated to their final destination before the corresponding metadata blocks are committed to the journal.

5.2.2 *Deployment Model*

All truncation, file deletion, and journaling operations can be expressed and performed as secure writes and deletions to data and metadata blocks. The resulting deployment model and its applicability are similar to those of a journaling layer. We inserted around 60 TAP-reporting calls in ext3 and jbd, with most collocated with block-layer interface write submission functions and various dirty functions (e.g., ext3_journal_dirty_data).

Applicable block types: Secure writes and deletions are performed for sensitive data, i-node, extended-attribute, indirect, and directory blocks, and corresponding structures written to the journal. Remaining metadata blocks (e.g., superblocks) are frequently updated and shared among files (e.g., bitmaps) and do not contain significant information about files. By not treating these blocks sensitively, we reduce the number of secure-deletion operations.

Secure data updates (Figure 2): Ext3/jbd calls TE_report_write() on sensitive data block updates, and TAP creates per-sector write entries. Updates to the same TAP write entries are consolidated via GUIDs; this behavior reflects that of the page cache.

The data update eventually reaches the block-layer interface (via commit), which retrieves the sensitive status via TE_check_info(). The layer can then perform the secure-write operation and invoke TE_cleanup_write() to remove the corresponding write entries.

Secure metadata updates: A metadata block must be securely written to and deleted from the journal. Ext3 reports pending journal writes to TAP via TE_report_copy().

Jbd manages its in-use persistent journal locations through its own superblock allocation pointers and a clean-up function, which can identify locations no longer in use. Through TE_report_delete(), we can put those locations on the deletion list and associate them with the journal superblock update. After the journal superblock is securely updated, the locations on the deletion list can be securely wiped. In the case of a crash, we securely delete all journal locations through TE_report_delete() once all committed updates have been securely applied.

Secure data deletions (Figure 3): When deleting sensitive file content, ext3's truncate function informs TAP of the deletion list and associated file i-node via TE_report_delete(). Given the transactional semantics of a journal (§5.2.1), we can associate the content-deletion event with the file's i-node update event instead of the free-block bitmap update event. Thus, we securely delete data before step ***(t1)***.

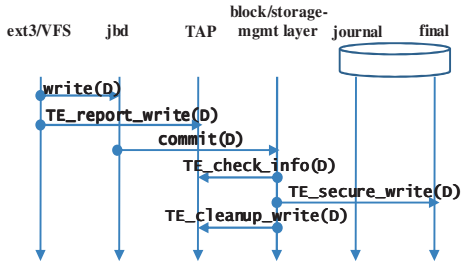


Figure 2. Secure data updates. D is the data block in various stages of being securely written.

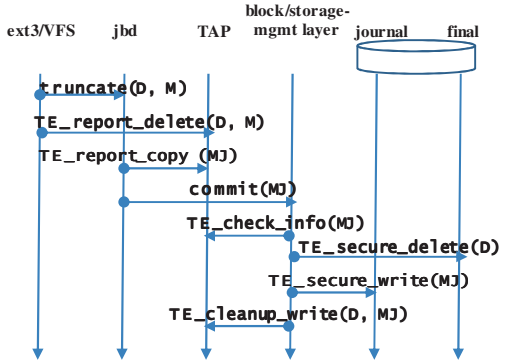


Figure 3. Secure data deletion. M is the updated metadata block; D is the data block in various stages of secure deletion; MJ is the metadata journal block that corresponds to the updated metadata block M.

TAP will create the i-node write entry and pair it with the corresponding secure-deletion reminder to hold the deletion list. When the write entry is copied via TE_report_copy(), reminders are transferred to the journal copy to ensure that secure deletions are applied to the matching instance of the i-node update.

When the block-layer interface receives the request to commit the update of the sensitive i-node to the journal, the interface calls TE_check_info() and retrieves the sensitive status of the i-node, along with the deletion list. The data areas are then securely deleted before the i-node update is securely written to the journal.

Secure metadata deletions: During a file truncation or deletion, ext3 also deallocates extended attribute block(s) and indirect block(s). Those blocks are attached to the i-node's list of secure-deletion reminders as well.

To securely delete an i-node or a file name in a directory, the block containing the entry is securely updated and reported via TE_report_write(). Additionally, we need to zero out the i-node and variable-length file name in the in-memory copies, so that they will not negate the secure write performed at the storage-management layer.

If a directory is deleted, its content blocks will be deleted in the same way as the content from a file.

Miscellaneous cases: Committed transactions might not be propagated instantly to their final locations. Across committed transactions, the same metadata entry (e.g., i-node) might have changed file ownership and sensitive status. Thus, jbd may consolidate, say, a non-sensitive update 1, sensitive update 2, and non-sensitive update 3 to the same location into a non-sensitive update. As a remedy, once a write entry is marked sensitive, it remains sensitive until securely written.

5.3 Enhanced FTL Storage-management Layer

We modified the existing Linux INFTL to incorporate secure deletion. INFTL uses a stack-based algorithm to remap logical pages to physical ones.

5.3.1 INFTL Extensions and Optimizations

INFTL remaps at the flash block level, where each 16-Kbyte flash block contains 32 512-byte pages, with a 16-byte control area per page. A remapped page always has the same offset within a block.

A NAND page can be in three states: empty, valid with data, or invalid. An empty page can be written, but an invalid page has to be erased to become an empty page.

***INFTL in-place updates*:** INFTL uses a stack of flash blocks to provide the illusion of in-place updates. When a page P1 is first written, an empty flash block B1 is allocated to hold P1. If P1 is written again (P1'), another empty flash block B2 is allocated and stacked on top of B1, with the same page offset holding P1'. Suppose we write P2, which is mapped to the same block. P2 will be stored in B2 because it is at the top of the stack, and its page at page offset for P2 is empty.

The stack will grow until the device becomes full; it will then be flattened into one block containing only the latest pages to free up space for garbage collection.

***INFTL reads*:** For a read, INFTL traverses down the appropriate stack from the top and returns the first valid page. If the first valid page is marked deleted, or if no data are found, INFTL will return a page of zeros.

Secure-deletion extensions: Our added secure write command is similar to the current INFTL in-place update. However, if a stack contains a sensitive page, we set its maximum depth to 1 (0 is the stack top). Once it reaches the maximum, the stack must be consolidated to depth 0. When consolidating, old blocks are immediately erased via the flash erase command, instead of being left behind.

Since the existing stack algorithm already tracks old versions, we also implemented the delayed-deletion optimization, which allows data blocks to defer the secure-write consolidation to file deletion time. Basically, the maximum depth is no longer bounded. Delaying secure deletion for metadata is trickier and will be investigated in future work.

A secure delete is a special case of a secure write. When a page is to be securely deleted, an empty flash block is allocated on top of the stack. All the valid pages, minus the page to be securely erased, are copied to the new block. The old block is then erased.

5.3.2 Disabled storage-management optimizations

Because jbd does not allow reordering to violate file system constraints and our flash has no built-in cache, we do not disable these optimizations.

6. VERIFICATION

We (1) tested the basic cases, assumptions, and corner cases discussed in §4.4 and (2) verified the state space of TAP.

6.1 Basic Cases

Sanity checks: We verified common cases of secure writes and deletes for empty, small, and large files and directories using random file names and sector-aligned content. After deletion, we scanned the raw storage and found no remnants of the sensitive information. We also traced common behaviors involving sensitive and non-sensitive objects; when the operation included a source and a destination, we tested all four possible combinations. The operations checked included moving objects to new directories, replacing objects, and making and updating symbolic and hard links. We also tested sparse files. In all cases, we verified that the operations behaved as expected.

Simulation of workload: We ran the PostMark benchmark [16] with default settings, modified with 20% of the files marked sensitive, with random content. Afterwards, we found no remnants of sensitive information.

Missing updates: To check that all update events and block types are reported, we looked for errors such as unanticipated block-type changes and unfound write entries in TAP, etc., which are signs of missing reports from the file system. Currently, all updates are reported.

Cases related to file-system-consistency properties: For cases derived from the reuse-ordering property, we created an ext3 file system with most of its i-nodes and blocks allocated, to encourage reuse. Then we performed tight append/truncate and file creation/deletion loops with alternating sensitive status. We used uniquely identifiable file content to detect sensitive information leaks and found none.

For pointer-ordering-related cases, we verified our ability to recover from basic failures and remove remnants of sensitive information. We also verified that the page cache prohibits unreferenced data blocks from being written to the storage.

Since the page ID part of GUIDs increases monotonically, we can use this property to detect illegal reordering of sensitive updates for the cases derived from the non-rollback property. For consolidations within a transaction, we used tight update loops with alternating sensitive modes. For consolidations across transactions, we used tight file creation/deletion loops with alternating sensitive modes. We checked all consolidation orderings for up to three requests (e.g., non-sensitive/sensitive/non-sensitive).

6.2 TAP Verification

We enumerated the TAP state-transition table and verified its correctness via two-version programming.

State representation: We exploited TAP's properties to trim the state space. First, a write entry will not consolidate with other write entries. This property ensures that each sensitive update is carried out unless explicitly cancelled. Various consolidation behaviors (e.g., page cache) are achieved by performing updates directly to the write entry. Second, the next state transition is based on current write entries of different types within a current state (plus inputs). With those two properties, we can reduce the representation of a state to at most one write entry of each type, and explore all state-generating rules.

To illustrate, each state holds one write entry for nine block types: data, i-node, other metadata, journal copy of data, journal copy of i-node, journal copy of other metadata, copy of data, copy of i-node, and copy of other metadata. Additionally, each write entry has four status bits: allocated, sensitive, having reminder attached, and ready to be deleted from the journal. Thus, a state is a 9x4 matrix and can be represented as 36 bits, with 2^{36} states.

State transitions: Each interface call triggers a state transition based on the input parameters. For example, the first TE_report_write() on a non-sensitive i-node will transition from the empty state (a zero matrix), say S0, to a state S1, where the allocated bit for the i-node is set to 1. If TE_report_write() is called again to mark the i-node as sensitive, S1 is transitioned to a new state S2, with allocated and sensitive bits set to 1s.

State-space enumeration: To enumerate states and transitions, we permuted all TAP interface calls with all possible input parameters to the same set of write entries. A small range of GUIDs was used so that each write entry could have a unique GUID, but GUID collisions were allowed to test error conditions. Given that the enumeration step can be viewed as traversing a state-space tree in breadth-first order, the tree fanout at each level is the total number of interface call-parameter combinations (261). As an optimization, we visited only states reachable from the starting empty state, and avoided repeated state-space and sub-tree branches. As a result, we explored a tree depth of 16 and located ~10K unique reachable states, or ~2.7M state transitions.

Two-version-programming verification: Given that the state-transition table is filled with mostly illegal transitions, we applied n-version programming to verify the table, where the probability of hitting the same bug with the same handling can be reduced as we add more versions. In this work, n=2. We wrote a user-level state-transition program based on hundreds of conceptual rules (e.g., marking a write entry of any type as sensitive will set the sensitive bit to 1). The enumerated state-transition table was reconciled with the one generated by the TAP kernel module.

7. PERFORMANCE EVALUATION

We compared TrueErase to an unmodified Linux 2.6.25.6 running ext3. We ran PostMark [16] to measure the overhead for metadata-intensive small-file I/Os. We also compiled OpenSSH version 5.1p1 [23] to measure the overhead for larger files. We ran our experiments on an Intel® Pentium® D CPU 2.80GHz dual-core Dell OptiPlex GX520 with 4-GB DDR533 and 1-GB DoC MD2203-D1024-V3-X 32-pin DIP mounted on a PCI-G DoC evaluation board. Each experiment was repeated 5 times. The 90% confidence intervals are within 22%.

PostMark: We used the default configuration with the following changes: 10K files, 10K transactions, 1-KB block size for reads and writes, and a read bias of 80%. We also modified PostMark to create and mark different percentages of files as sensitive. These files can be chosen randomly or with spatial locality, which is approximated by choosing the first x% of file numbers. Before running tests for each experimental setting, we dirtied our flash by running PostMark with 0% sensitive files enough times to trigger wear leveling. Thus, our experiments reflect a flash device operating at steady state. A `sync` command was issued after each run and is reflected in the elapsed time.

Table 3 shows that when TrueErase operates with no sensitive files, metadata tracking and queries account for 3% overhead compared to the base case. With 10% of files marked sensitive, the slowdown factor can be as high as 11, which confirmed the numbers in a prior study [41]. However, with 5% of files marked sensitive and with locality and delayed secure deletion of file data blocks, the slowdown factor can be reduced to 3.4, which is comparable to disk-based secure-deletion numbers [14].

We noticed some feedback amplification effects. Longer runs mean additional memory page flushes, which translate into more writes, which involve more reads and erases as well and lead to even longer running times. Thus, minor optimizations can improve performance significantly.

Table 3: Postmark flash operations, times, and overhead percentage compared to base.

	page reads	control-area reads	page writes	control-area writes	erases	time (secs)
Base	300K	1.97M	218K	237K	4.28K	671
0%	0.99x	1.08x	1.01x	1.01x	1.00x	**1.03x**
1% random	3.69x	2.09x	2.82x	2.79x	2.58x	2.93x
1% locality	2.95x	1.89x	2.33x	2.31x	2.16x	2.44x
1% random, delayed deletion	3.41x	2.00x	2.61x	2.59x	2.47x	2.73x
1% locality, delayed deletion	2.77x	1.77x	2.20x	2.19x	2.08x	2.29x
5% random	10.3x	4.22x	6.91x	6.83x	6.67x	7.39x
5% locality	6.69x	3.19x	4.86x	4.81x	4.32x	5.05x
5% random, delayed deletion	7.56x	3.48x	4.99x	4.99x	5.18x	5.54x
5% locality, delayed deletion	4.40x	2.33x	3.29x	3.29x	3.02x	**3.42x**
10% random	15.3x	5.82x	9.96x	9.84x	9.75x	**10.7x**
10% locality	9.96x	4.24x	7.00x	6.92x	6.23x	7.27x
10% random, delayed deletion	9.44x	4.23x	5.91x	5.96x	6.54x	6.80x
10% locality, delayed deletion	5.82x	2.96x	4.19x	4.22x	3.90x	4.45x

Table 4: Compilation flash operations, times, and overhead percentage compared to base.

	page reads	control-area reads	page writes	control-area writes	erases	time (secs)
make + sync						
Base	25.3K	108K	22.5K	23.9K	352	89
Random	4.79x	3.10x	3.15x	3.15x	3.13x	2.51x
Random, delayed deletion	1.70x	1.37x	1.41x	1.43x	1.40x	**1.41x**
make clean + sync						
Base	1.60K	3.73K	445	514	22	3
Random	10.0x	10.1x	13.6x	15.0x	7.14x	8.13x
Random, delayed deletion	8.47x	8.36x	11.0x	12.6x	6.22x	**6.87x**
Total						
Base	26.9K	112K	23.0K	24.4K	374	92
Random	5.10x	3.33x	3.35x	3.40x	3.37x	2.70x
Random, delayed deletion	2.10x	1.60x	1.59x	1.66x	1.69x	**1.59x**

OpenSSH compilations: We issued `make+sync` and `make clean+sync` to measure the elapsed times for compiling and cleaning OpenSSH [23]. For the TrueErase case, we marked the `openbsd-compat` directory sensitive before issuing `make`, which would cause all newly created files (e.g., .o files) in that directory to be treated sensitively. These files account for roughly 27% of the newly generated files (8.2% of the total number of files and 4.1% of the total number of bytes after compilation). Before running each set of tests, we dirtied the flash in the same manner as with PostMark and discarded the first run that warms up the page cache. Table 4 shows that a user would experience a compilation slowdown within 1.4x under the delay-deletion mode. A user would experience a slowdown within 6.9x under the delayed-deletion mode with a deletion-intensive workload. For the entire compilation cycle of `make+sync` with `make clean+sync`, a user would experience an overall slowdown within 60% under the delayed-deletion mode.

Overall, we find that the overhead is within our expectations. Further improvements in performance are future work.

8. RELATED WORK

This section discusses existing cross-layer secure-deletion solutions.

A semantically-smart-disk system (SDS) [36] observes disk requests and deduces common file-system-level information such as block types. The File-Aware Data-Erasing Disk is an ext2-based SDS that overwrites deleted files at the file-system layer.

A type-safe disk [33] directly expands the block-layer interface and the storage-management layer to perform free-space management. Using a type-safe disk, a modified file system can specify the allocation of blocks and their pointer relationships. As an example, this work implements secure deletion on ext2. Basically, when the last pointer to a block is removed, the block can be securely deleted before it is reused.

Lee et al. [18] have modified YAFFS, a log-structured file system for NAND flash, to handle secure file deletion. The modified YAFFS encrypts files and stores each file's key along with its metadata. Whenever a file is deleted, its key is erased, and the encrypted data blocks remain. Sun et al. [38] modified YAFFS and exploited certain types of NAND flash that allow overwriting of pages to achieve secure deletion. Raerdon et al. [27] also modified YAFFS to use a flash-chip-specific zero-overwriting technique. In addition, Raerdon et al. [26] developed the Data Node Encrypted File System (DNEFS), which modifies the flash file system UBIFS to perform secure deletion at the data node level, which is the smallest unit of reading/writing. DNEFS performs encryption and decryption of individual data nodes and relies on a key generation and deletion scheme to prevent access to overwritten or deleted data. Since UBIFS is designed for flash with scaling constraints, this approach is not as applicable for disks and larger-scale storage settings.

9. FUTURE WORK

Many opportunities exist to increase TrueErase's performance on NAND flash. We can implement flash-chip-specific zero-overwriting or scrubbing routines [27, 38, 41]. However, this optimization may make our solution less portable. We can add encryption to our system and use TrueErase to ensure secure deletion of the encryption key. We could also batch flash erasures for better flash performance.

Other future work will include tracking sensitive data between files and applications via tainting mechanisms, expanded handling of other threat models, and generalizations to handle swapping, hibernation, RAID, and volume managers.

10. LESSONS LEARNED/CONCLUSION

This paper presents our third version of TrueErase. Overall, we found that retrofitting security features to the legacy storage data path is more complex than we first expected.

Initially, we wanted to create a solution that would work with all popular file systems. However, we found the verification problem became much more tractable when working with file systems with proven consistency properties, as described in § 4.4.

Our earlier designs experimented with different methods to propagate information across storage layers, such as adding new special synchronous I/O requests and sending direct flash commands from the file system. After struggling to work against the asynchrony in the data path, we instead associated secure-deletion information with the legacy data path flow. We also decoupled the storage-specific secure-deletion action from the secure information propagation for ease of portability to different storage types.

We also found it tricky to design the GUID scheme due to in-transit versions and the placement of GUIDs. To illustrate, using only the sector number was insufficient when handling multiple in-transit updates to the same sector with conflicting sensitive statuses. Placing a GUID in transient data structures such as a block I/O structures led to complications when these structures could be split, concatenated, copied, and even destroyed before reaching storage. We solved this problem by associating a GUID with the specific memory pages that contain the data.

Tracking-granularity issues exist throughout the datapath. Data is stored in memory pages. File systems interact with blocks, multiples of which may exist on one memory page. The block layer may concatenate blocks together to form requests, which may span more than one memory page. Finally, requests are broken up into storage-specific granularities (e.g., flash pages). Metadata entries with mixed sensitive status can collocate within various access units as well. Various granularities make it difficult to map our solution to existing theoretical verification frameworks [34].

Finally, our work would not have been possible without direct access to a flash FTL. An unfortunate trend of FTLs is that they are mostly implemented in hardware, directly on the flash device controller. An implication is that most FTLs (and their wear-leveling/block-management routines) cannot be seen or accessed by the OS. To leave the door of software FTL research open, we need to create an environment that enables and eases experimentation, to demonstrate the benefits of software-level developments and controls.

To summarize, we have presented the design, implementation, evaluation, and verification of TrueErase, a legacy-compatible, per-file, secure-deletion framework that can stand alone or serve as a building block for encryption- and taint-based secure deletion solutions. We have identified and overcome the challenges of specifying and propagating information across storage layers. We show we can handle common system failures. We have verified TrueErase and its core logic via cases derived from file-system-consistency properties and state-space enumeration. Although a secure-deletion solution that can withstand diverse threats remains elusive, TrueErase is a promising step toward this goal.

11. ACKNOWLEDGMENTS

We thank Peter Reiher and anonymous reviewers for reviewing this paper. This work is sponsored by NSF CNS-0845672/CNS-1065127, DoE P200A060279, PEO, and FSU. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF, DoE, PEO, or FSU.

12. REFERENCES

[1] Bauer, S. and Priyantha, N.B. 2001. Secure data deletion for Linux file systems. *Proceedings of the 10th Usenix Security Symposium* (2001), 153–164.

[2] Boneh, D. and Lipton, R. 1996. A revocable backup system. *USENIX Security Symposium* (1996), 91–96.

[3] Cooke, J. 2007. Flash memory technology direction. *Micron Applications Engineering Document*. (2007).

[4] CWE - CWE-327: Use of a Broken or Risky Cryptographic Algorithm (2.2): *http://cwe.mitre.org/data/definitions/327.html*. Accessed: 2012-09-05.

[5] Diesburg, S.M., Meyers, C.R., Lary, D.M. and Wang, A.I.A. 2008. When cryptography meets storage. *Proceedings of the 4th ACM International Workshop on Storage Security and Survivability* (2008), 11–20.

[6] Ganger, G.R. 2001. *Blurring the line between OSes and storage devices*. Technical Report CMU-CS-01-166, Carnegie Mellon University.

[7] Garfinkel, S.L. and Shelat, A. 2003. Remembrance of data passed: a study of disk sanitization practices. *Security Privacy, IEEE*. 1, 1 (Feb. 2003), 17 – 27.

[8] Geambasu, R., Kohno, T., Levy, A.A. and Levy, H.M. 2009. Vanish: increasing data privacy with self-destructing data. *Proceedings of the 18th USENIX Security Symposium* (Berkeley, CA, USA, 2009), 299–316.

[9] Halderman, J.A., Schoen, S.D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J.A., Feldman, A.J., Appelbaum, J. and Felten, E.W. 2009. Lest we remember: cold-boot attacks on encryption keys. *Commun. ACM*. 52, 5 (May. 2009), 91–98.

[10] Health Insurance Portability and Accountability Act of 1996: *http://www.hhs.gov/ocr/privacy/hipaa/administrative/statute/hipaastatutepdf.pdf*. Accessed: 2012-07-24.

[11] Hughes, G. 2004. *CMRR Protocols for disk drive secure erase*. Technical report, Center for Magnetic Recording Research, University of California, San Diego.

[12] Hughes, G.F. 2002. Wise drives [hard disk drive]. *Spectrum, IEEE*. 39, 8 (Aug. 2002), 37 – 41.

[13] Ironkey: *http://www.ironkey.com*. Accessed: 2012-07-26.

[14] Joukov, N., Papaxenopoulos, H. and Zadok, E. 2006. Secure deletion myths, issues, and solutions. *Proceedings of the Second ACM Workshop on Storage Security and Survivability* (New York, NY, USA, 2006), 61–66.

[15] Joukov, N. and Zadok, E. 2005. Adding secure deletion to your favorite file system. *Security in Storage Workshop, 2005. SISW '05. Third IEEE International* (Dec. 2005), 8 pp.–70.

[16] Katcher, J. 1997. *Postmark: A new file system benchmark*. Technical Report TR3022, Network Appliance, 1997. www. netapp. com/tech_library/3022. html.

[17] King, C. and Vidas, T. 2011. Empirical analysis of solid state disk data retention when used with contemporary operating systems. *Digital Investigation*. 8, (2011), S111–S117.

[18] Lee, J., Heo, J., Cho, Y., Hong, J. and Shin, S.Y. 2008. Secure deletion for NAND flash file system. *Proceedings of the 2008 ACM Symposium on Applied Computing* (New York, NY, USA, 2008), 1710–1714.

[19] Mac OS X Security Configuration for Mac OS X Version 10.6 Snow Leopard: *http://images.apple.com/support/security/guides/docs/SnowLeopard_Security_Config_v10.6.pdf*. Accessed: 2012-07-25.

[20] Marcel Breeuwsma, Martien De Jongh, Coert Klaver, Ronald Van Der Knijff and Roeloffs, M. 2009. *Forensic Data Recovery from Flash Memory*. CiteSeerX.

[21] National Industrial Security Program Operating Manual 5220.22-M: 1995. *http://www.usaid.gov/policy/ads/500/d522022m.pdf*. Accessed: 2012-07-26.

[22] Nightingale, E.B., Veeraraghavan, K., Chen, P.M. and Flinn, J. 2008. Rethink the sync. *ACM Trans. Comput. Syst.* 26, 3 (Sep. 2008), 6:1–6:26.

[23] OpenSSH: *http://openssh.com/*. Accessed: 2012-06-07.

[24] Perlman, R. 2005. *The ephemerizer: making data disappear*. Sun Microsystems, Inc.

[25] Peterson, Z.N.J., Burns, R., Herring, J., Stubblefield, A. and Rubin, A. 2005. Secure deletion for a versioning file system. *Proceedings of the USENIX Conference on File And Storage Technologies (FAST)* (2005), 143–154.

[26] Reardon, J., Capkun, S. and Basin, D. 2012. Data Node Encrypted File System: Efficient Secure Deletion for Flash Memory. *21st USENIX Security Symposium* (Aug. 2012).

[27] Reardon, J., Marforio, C., Capkun, S. and Basin, D. 2011. *Secure Deletion on Log-structured File Systems*. Technical Report arXiv:1106.0917.

[28] Scrub utility: *http://code.google.com/p/diskscrub/*. Accessed: 2012-07-26.

[29] Secure rm: *http://sourceforge.net/projects/srm/*. Accessed: 2012-07-26.

[30] Secure USB Flash Drives | Kingston: *http://www.kingston.com/us/usb/encrypted_security*. Accessed: 2012-07-26.

[31] shred(1) - Linux man page: *http://linux.die.net/man/1/shred*. Accessed: 2012-08-13.

[32] Shu, F. and Obr, N. 2007. *Data set management commands proposal for ATA8-ACS2*.

[33] Sivathanu, G., Sundararaman, S. and Zadok, E. 2006. Type-safe disks. *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2006), 15–28.

[34] Sivathanu, M., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H. and Jha, S. 2005. A logic of file systems. *Proceedings of the 4th USENIX Conference on File and Storage Technologies - Volume 4* (Berkeley, CA, USA, 2005), 1–1.

[35] Sivathanu, M., Bairavasundaram, L.N., Arpaci-Dusseau, A.C. and Arpaci-Dusseau, R.H. 2004. Life or death at block-level. *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6* (Berkeley, CA, USA, 2004), 26–26.

[36] Sivathanu, M., Prabhakaran, V., Popovici, F.I., Denehy, T.E., Arpaci-Dusseau, A.C. and Arpaci-Dusseau, R.H. 2003. Semantically-smart disk systems. *Proceedings of the 2nd USENIX Conference on File and Storage Technologies* (2003), 73–88.

[37] Special Publication 800-88: Guidelines for Media Sanitization: 2006. *http://csrc.nist.gov/publications/nistpubs/800-88/NISTSP800-88_with-errata.pdf*. Accessed: 2012-07-26.

[38] Sun, K., Choi, J., Lee, D. and Noh, S.H. 2008. Models and Design of an Adaptive Hybrid Scheme for Secure Deletion of Data in Consumer Electronics. *Consumer Electronics, IEEE Transactions on*. 54, 1 (Feb. 2008), 100 –104.

[39] The OpenSSD Project: *http://www.openssd-project.org/wiki/The_OpenSSD_Project*. Accessed: 2012-07-29.

[40] Thibadeau, R. 2006. Trusted Computing for Disk Drives and Other Peripherals. *Security Privacy, IEEE*. 4, 5 (Oct. 2006), 26 –33.

[41] Wei, M., Grupp, L.M., Spada, F.E. and Swanson, S. 2011. Reliably erasing data from flash-based solid state drives. *Proceedings of the 9th USENIX Conference on File and Stroage Technologies* (Berkeley, CA, USA, 2011), 8–8.

[42] Wipe: Secure File Deletion: *http://wipe.sourceforge.net/*. Accessed: 2012-07-26.