Biography: Distinguished Practitioner

Richard Kemmerer
University of California, Santa Barbara

Richard A. Kemmerer is the Computer Science Leadership Professor and a past Department Chair of the Department of Computer Science at the University of California, Santa Barbara. Dr. Kemmerer received the B.S. degree in Mathematics from the Pennsylvania State University in 1966, and the M.S. and Ph.D. degrees in Computer Science from the University of California, Los Angeles, in 1976 and 1979, respectively. His research interests include formal specification and verification of systems, computer system security and reliability, programming and specification language design, and software engineering. He is author of the book, Formal Specification and Verification of an Operating System Security Kernel, and a co-author of Computers at Risk: Safe Computing in the Information Age, For the Record: Protecting Electronic Health Information, and Realizing the Potential of C4I: Fundamental Challenges. Dr. Kemmerer is a Fellow of the IEEE Computer Society, a Fellow of the Association for Computing Machinery, a member of the IFIP Working Group 11.3 on Database Security, and a member of the International Association for Cryptologic Research. He is a past Editor-in-Chief of IEEE Transactions on Software Engineering and has served on the editorial boards of the ACM Computing Surveys and IEEE Security and Privacy. He currently serves on the Board of Governors of the IEEE Computer Society and on Microsoft's Trustworthy Computing Academic Advisory Board.

So You Think You Can Dance?

Richard A. Kemmerer
Computer Security Group
Computer Science Department
University of California
Santa Barbara, CA 93106
kemm@cs.ucsb.edu

Abstract

This paper discusses the importance of keeping practitioners in mind when determining what research to pursue and when making design and implementation decisions as part of a research program. I will discuss how my 30 plus years of security research have been driven by the desire to provide products, tools, and techniques that are useful for practitioners. I will also discuss my view of what new security challenges the future has in store for us.

1. Introduction

I am really thrilled that you chose me for the Distinguished Practitioner Award. Throughout my research career I have always tried to work on problems whose solution would be useful to others. My approach to research has been one of not just developing new theories, but rather to take the theoretical ideas that I have produced and reduce them to practice. I have always wanted my developments to be used by practitioners. Whenever possible I have tried to provide tools and techniques that others could use to solve the same problems on similar systems or to solve similar problems.

Unfortunately, I do not think that my view is shared by many academic researchers. It is more common for academic researchers to ask "What research can I pursue that will give me a quick route to publishable papers?" or "What can I do to get more funding?". My approach to research is likely the result of having spent eight years in the trenches as a programmer, system designer, and IT manager before pursuing my PhD and jumping to the academic side.

I would like to say a few words about why I think it is important to keep the practitioners in mind when determining what research to pursue and when making design and implementation decisions as part of a research program. In my view, the goal of every research project should be that it ends up in use. That is, the research should produce a useful technique, a useful tool, a useful product, or a combination of these three. A question that may be raised is useful to who? For instance, an academic researcher might consider their research effort to have produced a useful product if it resulted in a published paper. I believe that the touchstone here is to ask would a practitioner use my technique, tool, or product? As researchers we should strive to make sure that practitioners are aware of our work (ACSAC is a good venue for this), that our work is relevant to the practitioner (will it make their life easier?), and will the practitioner be able to use it (does it require a special expertise that most practitioners do not have?).

In the next section I will discuss how my 30 plus years of security research have been driven by the desire to provide products, tools, and techniques that are useful for practitioners. At the end of each section I will point out what result (i.e., technique, tool, or product) was produced. I will also try to point out some lessons that I learned along the way. In section 3, I will discuss my view of what new security challenges the future has in store for us.

2. Examples From Thirty Years of Experience

2.1. Data Secure Unix

Thirty some years ago I started working in Computer Security. My path to get here was different from most. I was primarily interested in Formal Methods and I was looking around for a real application that needed a high level of assurance. At this time most of the formal methods researchers were proving properties about small programs, such as sorting algorithms. Although I believe that having a sorting algorithm work correctly is a good idea, I was looking for something more useful and large. I chose the Data Secure Unix kernel as my application. Data Secure

Unix was a kernel structured operating system, which was constructed as part of an ongoing effort at UCLA to develop procedures by which operating systems could be produced and shown to be secure [48]. Furthermore, because this was a real operating system kernel, I felt that it was important to get a correct statement of what security properties were being proved.

The proof that the kernel is secure had two parts: a) developing four levels of specifications, ranging from the top-level security criterion to the Pascal code, and b) verifying that the specifications at these different levels of abstraction were consistent with one another. The top-level formalized our intuitive notion of data security for the kernel. The next level contained operations corresponding to uninterruptible kernel calls and more specific objects like pages, processes, and devices. The low-level specification, which used data structures from the implementation, were still at a higher level than the code because detail was omitted and some abstracting of the data structures was employed. The lowest level was the actual code.

When starting this work, I looked around and found that Wulf, London and Shaw had developed a refinement technique for their object oriented language Alphard [49, 41]. Their methodology provided a formal framework to prove that abstract specifications of an abstract data type are consistent with the code that implements that type. Their refinement techniques allowed one to go from a high-level specification to code. I extended and altered this work to allow for multiple levels of specification, and eventually code.

This experience addressed all aspects of specification and verification in a significant software project. That is, we verified a large-scale, production level software system, including all aspects from initial specification to verification of the implemented code. The result of this work was a refinement *technique* for formally specifying and verifying large multi-level specifications down to the implemented code. The proofs were all done by hand, using a derivation approach presented by Kalish and Montague [22]. Next, the AFFIRM verification system [14] at ISI was employed as a proof checker to verify the proofs. The full details of the verification can be found in [48].

Result: Technique

Lesson learned: *Stand on the shoulders of others, not their toes.*

2.2. Covert Channels

Secure computer systems use both mandatory and discretionary access controls to restrict the flow of information through legitimate communication channels, such as files, shared memory, and process signals. Unfortunately, in practice one finds that computer systems are built such that users are not limited to communicating only through the intended communication channels. As a result, a well-founded concern of security-conscious system designers is the potential exploitation of system storage locations and timing facilities to provide unforeseen communication channels to users. These illegitimate channels are known as *covert storage* and *covert timing channels*. The idea is that covert channels use entities not normally viewed as data objects, such as file locks, device busy flags, and the passing of time, to transfer information from one subject to another.

I was introduced to covert channels early in my academic career. I was consulting for Systems Development Corporation (SDC) in Santa Monica and they were subcontracting to a company called Digital Technology Incorporated (DTI) in Champaign, Illinois. The product that DTI was developing was a secure network front-end for the AUTODIN II network [17]. One of SDC's tasks was to perform a covert channel analysis on the network front-end, and I was asked to lead this project. One of the first questions that I asked was "What method did people use to discover covert channels in the past?". The response I got was "Well we found this channel on system XX or that channel on system YY." I quickly concluded that there was no approach; everything was *ad hoc*. The closest thing to an approach was to see if a channel that was known to exist in one system also occurred in the system being analyzed, such as using the lock bits on a file to signal information. The second question I asked was "How do you know when you are done?". Again I was greeted with a lot of blank stares. One thing that was perfectly clear to me was that we needed something more rigorous than the *ad hoc* approach that was currently being used.

The Shared Resource Matrix (SRM) methodology is the approach that we developed to locate covert channels. This approach can be applied to a variety of system description forms and can increase the assurance that all channels have been found. It is easily reviewed, disregards resources that are not shared, and is iterative as the design is refined or changed. It can be used in all phases of the software life cycle on systems whose constituent parts are in varying phases of development.

Storage and timing channel analysis is performed in two steps in the Shared Resource Matrix methodology. First, all shared resources that can be referenced or modified by a subject are enumerated, and then each resource is carefully examined to determine whether it can be used to transfer information from one subject to another covertly. The methodology assumes that the subjects of the system are processes and that there is a single processor which is shared by all of the processes. The processes may be local or distributed; however, only one process may be active at any one time.

The key idea is that the nondata objects that are used for covert channels are resources that are needed to regis-

ter the state of the system. A shared resource is any object or collection of objects that may be referenced or modified by more than one process. The Shared Resource Matrix is a way of displaying the attributes of each resource and the operations that can manipulate them. Attributes of all shared resources are indicated in row headings of the matrix, and the operation primitives of the system make up the column headings. After determining all of the row and column headings one must determine for each attribute (the row headings) whether the primitive indicated by the column heading modifies or references that attribute. The generated matrix is then used to determine whether any channels exist. For a storage channel the sending process alters a particular data item, and the receiving process detects and interprets the value of the altered data to receive information covertly. With a timing channel the sending process modulates the amount of time required for the receiving process to perform a task or to detect a change in an attribute, and the receiving process interprets this delay or lack of delay as information.

The result of this research was a *technique* for identify covert channels, the SRM. The system that the technique was applied to was DTI's Communications Operating System Network Front End (COS/NFE) [17]. Clearly my example system for testing the SRM approach was a real system. Unfortunately, in addition to being real, it was *proprietary*. Proprietary systems pose a problem for academics, who are rewarded for publishing papers. As I was developing the SRM approach and applying it to the COS/NFE I was aware that as long as the COS/NFE was proprietary, I could not publish the details of my work. When I expressed my concern to the DTI management, I was told that it would only be proprietary for another six months. After two years of waiting and the system remaining proprietary, I finally wrote a paper presenting the SRM and showing the results of applying it to a toy problem. The journal that I submitted it to rejected the paper saying that it would never work for a real system, even though I mentioned the name of the real system and gave baud rates for the worst case covert channels that were found.

Result: Technique

Lesson learned: *Real systems are often proprietary or classified. This could be counter productive when trying to get tenure.*

Postscript: Four years later I got the paper published in *ACM Transactions on Computer Systems* [27] by expounding on the fact that even though the example used was a toy it had the properties of a real system and again pointing to the real system (still proprietary at that time) that was evaluated.

2.2.1 Honeywell Secure Ada Target

In 1986 both the Shared Resource Matrix (SRM) approach and the noninterference approach [16] were applied to a high level design for a real system – the Honeywell Secure Ada Target (SAT). The SAT was intended to meet or exceed all of the requirements for A1 certification. A formal model of the SAT was expressed in the GYPSY formal specification language. Both techniques were applied to the Gypsy abstract model of the SAT. In this case, values for the initial Shared Resource Matrix were provided by the Gypsy flow analyzer, which analyzed the specification and/or code. Each operation in the matrix corresponded to a Gypsy function or procedure, and the flow analyzer determined what components of the program's security state were read and/or modified by each operation. For the noninterference approach the failed proofs of the unwinding theorems lead the analyst to the flows to consider, but, like the Shared Resource Matrix approach, it too did not aid the analyst in the actual analysis. With both approaches the analyst had to come up with the signaling sequences and determine whether they could be used as covert channels. Both methods were successful in detecting covert channels. A detailed discussion of the application of both techniques and the nature of the covert channels discovered can be found in [18].

Result:

Lesson learned: *Don't be afraid to run a head–to–head experiment with your technique or tool. This is how you get credibility.*

2.2.2 Covert Flow Trees

As mentioned above, the SRM approach and the noninterference approach were used to determine what attributes might be used for signaling information. Neither approach produced the sequence of operations necessary to covertly signal the information. In 1991 Phil Porras and I introduced the idea of using tree data structures to model the flow of information from one shared attribute to another [23]. These trees were called Covert Flow Trees (CFTs). CFTs are used to perform systematic searches for operation sequences that allow information to be relayed through attributes and that are eventually detected by a listening process. When traversed, the paths of a CFT yield a comprehensive list of operation sequences that support communication via a particular resource attribute. These operation sequences are then analyzed and either discharged as benign or determined to be covert communication channels. That is, the analyst with his/her experience is still the one that makes the determination. The Covert Flow Tree tool, which was an implementation of this approach was built and distributed.

Result: Tool

Lesson learned: *It is beneficial to take a fresh look at an area.*

2.2.3 A modular approach to using the SRM

In 1996 Data General Corporation decided that they wanted their trusted DG/UX operating system to undergo a B2 evaluation. DG/UX was a full commercial-strength UNIX system with many features and support for a wide range of devices. The part of the system that was to be analyzed for covert channels was on the order of 800,000 lines of C code. The kernel was structured so that each of the elements of the system state was under the control of a single subsystem. That is, these elements could only be referenced or modified by functions of the controlling subsystem; thus, each subsystem could be thought of as an abstract object.

In order to make the covert channel analysis task for the Trusted DG/UX kernel more manageable and, in particular, to deal with the Ratings Maintenance Program (RAMP), a modular approach that takes advantage of the subsystem architecture was developed. The approach leveraged the subsystem architecture of the DG/UX kernel. First, an SRM analysis was performed on each of the subsystems that contained an exported function directly invoked from one of the system calls. These subsystems were called "peer subsystems." The information from the SRMs for all of the peer subsystems was then used to build a kernel-wide SRM.

There are two major advantages to this modular approach to covert channel analysis. The first is that the covert channel analysis can be decomposed into a number of separable tasks, which can be distributed among many developers. The second is that the system can be incrementally reanalyzed as it changes over time. The details of this work can be found in [24].

Result: Technique

Lesson learned: *Necessity is the mother of invention.*

2.3. UNISEX Symbolic Execution Tool

Symbolic execution is an approach that lets algebraic symbols represent the input values, rather than using numeric or character inputs. These symbols are then manipulated by the program. By placing restrictions on the values that each symbol may represent the symbolic input represents a class of input values rather than a single value. That is, by properly restricting the input values each symbolic execution of a program can correspond to executing the program on a particular subdomain. When execution is completed the symbolic values obtained by the variables are analyzed to determine if they meet the programmer's expectations. This method of testing is known as *symbolic execution*. A symbolic executor can be used with branch or path testing to find the restrictions that must be placed on input values to cause a particular path to be executed. A symbolic executor can also be used to generate the necessary verification conditions that need to be proved to verify that a formally specified program is consistent with its entry and exit specifications. The type of correctness being verified is partial correctness as originally defined by [15]. That is, if the entry assertion is true when the program is invoked and the program terminates, then the exit assertion will be true when the program halts. The UNISEX system is used in all three of these ways to test and verify Pascal programs.

This work was motivated by having used someone else's symbolic execution system for annotating programs and proving properties about the programs. The tool that I originally used was a system called Effigy, which was developed by Jim King at IBM Watson Labs [34]. Effigy used a PL/1-like toy language and ran only on IBM mainframe systems. There were two problems with this: the students were not familiar with the language, and they were not familiar with the IBM centralized system. This was particularly bad because to access the system remotely you had to go through a terminal server. Effigy was an interactive system that took the annotated program, compiled it into interactive code, and then interpretted the user's commands to symbolically execute the program. Unfortunately, it was not very robust. If a user mistyped a command or command argument, the system got confused and hung. The only way that the system could be reset was by physically disconnecting the terminal line. To make things worse, after disconnecting the terminal it was necessary to wait approximately five minutes to reconnect to the terminal server again.

Clearly this was not a satisfactory situation. Therefore, I decided to build my own symbolic execution system. The primary goal of the project was to have a testing and formal verification tool that was useful, particularly for students. This was the reason for choosing Pascal, which was the first programming language being taught at most universities at that time. Another goal was to have a system that could be used without change at a variety of educational and research facilities. By implementing the system on UNIX all of the UNIX tools (eg. editors and pretty printers) were immediately available.

The result was UNISEX, which is a UNIX-based symbolic executor for Pascal. The UNISEX system provides an environment for both testing and formally verifying Pascal programs. The system supports a large subset of Pascal, runs on UNIX-like systems and provides the user with a variety of debugging features to help in the difficult task of program validation. The system provides the user with system features that can be enabled or disabled at will making the system more useful. By allowing the user to control the amount of information provided by the system during a symbolic execution, the user is neither left in the dark nor inundated with more information than desired. The various display commands also make the system more user friendly. The details of UNISEX can be found at [32].

Result: Tool

Lesson learned: *Rather than just complain about tools*

that do not meet your needs, learn from their shortcomings and build what you need

2.3.1 Ada Symbolic Executor

Several years later when Ada was gaining in popularity, we developed a symbolic execution approach for executing concurrent programs written in the Ada programming language. Extending symbolic execution to concurrent programs is non-trivial, because concurrent programs are nondeterministic, and nothing can be assumed about the relative execution speeds of the concurrent execution units. Our symbolic execution model considered all relevant interleavings of concurrent execution units [19, 8]. We implemented a UNISEX-like symbolic executor based on the interleaving approach. One of the main concerns with this approach is the large number of states that could be produced. To address this concern, we developed methods for pruning duplicate paths. It should be noted, however, that although there are many paths to be considered the actual process of carrying out the symbolic execution and the corresponding proof of the generated verification conditions is not complex. It is an error prone job for humans, but is perfectly suited for automation.

Result: Technique, Tool

Lesson learned: *It is necessary to stay current.*

2.4. Testing Formal Specifications

My testing formal specifications work is something that was motivated by my experiences as a specification writer for real systems. It was my experience that when writing a high-level specification I would ask myself "what if" questions. For instance, I did this when working on the Secure Release Terminal (SRT) project [20] at SDC. Consider the case where a user is reviewing a document that is wider than the display screen and, therefore, requires moving the screen to the right or left over the document. Furthermore, assume that the screen is on the far right side of the document (some text on the left is hidden) and the user asks to scroll down. Should the screen display the part of the document directly below its current position without moving to the far left, or should it display the far left portion of the lines? I believe the desired result differs based on whether the user is viewing text documents or CAD diagrams. The important thing is that if the deign constrains the implementation to one of the two results and the customer wants the other, then there is a problem.

More generally, the problem is that although the specification satisfies the correctness criteria, there may be no implementation that is consistent with the specification and at the same time provides the desired functionality. The real problem is that this is usually not discovered until the design has gone through several levels of refinement, with each level being formally verified, and the implementation is in progress or completed. The result is a "yo-yo" effect where the designer goes back to the top level and rewrites the specification to allow an implementation that provides the desired functionality while preserving the correctness criteria.

This "yo-yo" effect is costly and time consuming, particularly where proofs have to be redone, because the specification has changed. An approach to reducing the "yo-yo" effect is to test the specifications to see if they allow the desired functionality, particularly for special cases. For instance, one could test what the result of performing a particular sequence of operations would be when starting in a particular configuration. This can be achieved by executing some test cases to see if the desired results are obtained. The problem is that most specification languages are nonprocedural. My initial approach to this problem was to use a pencil and paper to calculate the result of executing the specifications.

In [28] I presented two approaches to solving this problem. The first is to convert the nonprocedural specifications into a procedural form. This procedural form then serves as a rapid prototype to use for testing. The other approach is to perform a symbolic execution of the sequence of operations and check the resultant symbolic values to see if they define the desired set of resultant states. The paper discusses the advantages and disadvantages of each approach.

I decided to implement the symbolic execution approach, because I thought it was the more usable approach and I could build on my experience in building the UNISEX symbolic executor. The symbolic execution tool that we built was called Inatest [12]. The formal specification language processed by the tool was Ina Jo, which is a nonprocedural assertion language that is an extension of first-order predicate calculus. The language assumes that the system is modeled as a state machine. A complete description of the Ina Jo language can be found in the Ina Jo Reference Manual [37].

Result: Technique, Tool

Lesson learned: *Many good research projects rise out of problems that are encountered when working on what appears to be an unrelated project.*

Build on your previous work.

2.5. Verification Assessment Study

In 1984 the National Computer Security Center asked me to lead a study that was to determine the state-of-the-art of formal verification systems. The study began in November 1984 and lasted for approximately nine months. The main goal of this effort was a technology interchange among the developers of four established verification systems. The

systems investigated were i) Affirm (General Electric Company, Schenectady, New York), ii) FDM (System Development Corporation, Santa Monica, California) iii) Gypsy (the University of Texas at Austin, Austin, Texas), and iv) Enhanced HDM (SRI International, Menlo Park, California). There was some comparative work on examples, but the main idea was for the developers to learn the details of each other's system as a basis for future development.

The approach taken for this study was first to select a suitable set of example problems to be used to investigate the established systems. Each of the systems in turn was used to specify and verify these problems. The specification and verification was performed by the development team for each system. One member of each system's development team was picked as the "representative" for that particular system. The system representatives were well established with regard to their in-depth knowledge of the particular verification system. In most cases the representative was one of the original developers of the system.

For each of the four systems, the specification and verification of the example problems was done by the development team for that particular system. The nonresident members of the assessment group then visited the home site of each system to study the system and the solutions to the problems. During the site visits, each participant was allowed to study the system in any way he or she wished. Usually, this meant that the participant defined a favorite problem and investigated the effects that the system had on the development of a solution. For example, I worked with a secure release terminal example [20], on each of the four systems.

Through this technical interchange, members of both the assessment group and the system development teams were able to learn about each other's system and to use this knowledge as a basis for future development. After visiting a site, each of the nonresident participants prepared a critique of that particular system.

After all the site visits had been completed, the assessment team convened at the University of California in Santa Barbara, California, to compare their findings, to discuss the relevant verification technology issues that were raised during the study, and to propose future directions for verification research. The final report for this project was five volumes – one for each system and an executive overview [29].

After the project was done and all the reports were ready to be copied and distributed, the sponsoring agency offered to copy the report and mail it to our list of recipients. The report was five volumes and totaled more then 1400 pages, and I was on sabbatical at the time. If I were to do the copying and mailing myself, I would have to schlep the reports to Kinkos to be copied and to the Post Office to be mailed. Therefore, even though I had the funds in the contract to do this, I accepted what appeared to be a kind offer.

Little did I know that the report would be marked "restricted by the Arms Export Control Act." More than half of the 150 names that I had on my distribution list were unable to receive the report, because they were foreign nationals. This was in spite of the fact that my contract had no publishing restrictions and I did not need prior approval before publishing. Needless to say, this was not a good situation. I worked for more than two years to try and get the report released, but with no success.

Result:

Lesson learned: *Always look a gift horse in the mouth.*

2.6. ASLAN Formal Specification Language

One of the benefits of working on the verification assessment study was that I had formulated a good idea of what I wanted in a formal specification language. Previously when publishing papers on my work with SDC's Ina Jo specification language, I would change the syntax to be more readable. This variation of the Ina Jo syntax was what I used as the basis for my specification language, called ASLAN.

The ASLAN specification language is built on first order predicate calculus with equality and employs the state machine approach to specification. The system being specified is thought of as being in various states, depending on the values of the state variables. Changes in state variables take place only via well defined transitions. Predicate calculus is used to make assertions about desired properties that must hold at every state and between two consecutive states. Critical requirements that must be met in every state are state invariants.

To prove that a specification satisfies some invariant assertion, ASLAN generates the candidate lemmas needed to construct an inductive proof of the correctness of the specification with respect to the invariant assertion. These lemmas are known as correctness conjectures.

An ASLAN system specification is a sequence of levels. Each level is an abstract data type view of the system being specified. The first ("top level") view is a very abstract model of what constitutes the system (types, constants, variables), what the system does (i.e., state transitions), and the critical requirements the system must meet. Lower levels are increasingly more detailed. The lowest level corresponds fairly closely to high level code. ASLAN generates correctness conjectures whose proofs ensure that lower levels correctly refine upper levels. A more detailed overview of ASLAN can be found in [2] and the details of the language and the proof obligations can be found in [1].

The motivation for defining and implementing the ASLAN language and the ASLAN specification processor was much the same as the motivation for building UNISEX. That is, I wanted a tool that my students could use to learn how to write formal specifications and prove critical proper-

ties about the system being specified. I took the good things that I learned from different languages that I had used and put them in ASLAN. I also implemented the ASLAN specification processor on the Unix operating system.

The ASLAN system was used by the Microelectronics and Computer Technology Corporation (MCC), which was a computer industry research and development consortium. They used ASLAN as a central component of their Spectra software development environment. My view was, and still is, that by making my systems widely available other researchers and practitioners can easily experiment and gain first-hand experience with the system's capabilities. Therefore, when MCC approached me about using the ASLAN system, I was more than happy to let them have it (free of charge). I was not so happy, however, when I asked if I could get a copy of their Spectra software development environment. They told me that only members of the consortium could get access to Spectra.

Result: Technique, Tool

Lesson learned: *Always read the fine print.*

2.7. Analyzing Encryption Protocols

When considering a secure network that uses encryption to achieve its security one must consider both encryption algorithms and encryption protocols. An encryption algorithm, such as DES or RSA, is used to convert clear text into cipher text or cipher text into clear text. An *encryption protocol* is a set of rules or procedures for using the encryption algorithm to send and receive messages in a secure manner over a network.

In 1982 while listening to Manny Blum present his oblivious transfer protocol at CRYPTO 82, I had an epiphany. My thought was that by specifying encryption protocols formally and by also specifying their critical requirements formally one could generate the necessary proof obligations to assure that they meet their desired critical properties. More specifically, the idea is to specify formally the components of the cryptographic facility and the associated cryptographic operations. The components are represented as state constants and variables, and the operations as state invariants, and the theorems that must be proved to guarantee that the system satisfies the invariants are automatically generated by the verification system.

The approach does not attempt to prove anything about the strength of the encryption algorithms being used. On the contrary, it may assume that the obvious desirable properties, such as that no key will coincidentally decrypt text encrypted using a different key, hold for the encryption scheme being employed.

When I first tried my idea on some encryption protocols using Ina Jo for my specification language, I found that the amount of specification that was necessary to specify assumptions about the the encryption algorithms was so large that it made it unreasonable to work with the overall specification. I finally came upon a protocol that was used in a single-domain communication system using dynamically generated primary keys and two secret master keys, which was originally released by IBM with a single master key and was later recalled to add a second master key, because of a flaw in the protocol. The details of the system and the protocol can be found in [38].

What I found when attempting to prove that this protocol satisfied its critical requirements was that the proofs failed. However, I found that analyzing the failed proofs gave insight into errors or omissions in the specification or to flaws in the protocol itself. This experience showed that the approach was useful for both confirming previously known flaws and for discovering new flaws. The details of this initial experience can be found in [30, 31].

In 1990, I along with Cathy Meadows and John Millen (both of whom by this time were also working on analyzing encryption protocols) were asked by Gus Simmons to look at a mobile communication protocol, which was about to be patented. Gus had already found an error in the protocol, and he was challenging us to see if we could find it with our respective systems. By this time I was using an ASLAN-based symbolic execution system, called Aslantest [11], to test my encryption protocol specifications. The result was that I not only found the same error as Gus, but I also found some previously unknown flaws. Meadows and Millen also found new flaws with their systems [33].

More recently, I have been using the ASTRAL model checker, discussed below in section 2.8, to analyze protocols with specific real-time constraints [7]. This work has revealed a number of flaws and has helped to better understand these complex protocols.

Result: Technique, Tool

Lesson learned: *If at first you don't succeed, then keep on trying.*

Reuse is a good thing. Build on your own work.

2.8. ASTRAL

As discussed in section 2.6, the ASLAN language is for specifying and verifying sequential software systems. In the early 1990s, I decided to look at *formally verifying real-time systems*. I designed a specification language called ASTRAL, which was an extension of ASLAN with timing constraints. The ASTRAL language allows one to build modularized specifications of complex systems with layering. I also developed a formal proof system for proving critical properties for these systems.

In ASTRAL a real-time system is modeled by a collection of process specifications and a single global specification [3]. Each process specification is much like the

ASLAN specifications discussed in section 2.6. The ASTRAL proofs are divided into two categories: intra-level proofs and inter-level proofs [5, 6]. The former deals with proving that the specification of level i is consistent and satisfies the stated critical requirements, while the latter deals with proving that the specification of level i+1 is consistent with the specification of level i. Two or more ASTRAL system specifications can also be composed into a single specification of a combined system [4]. A proof system that generates the necessary proof obligations for the combined system was also developed.

Again, because I wanted the system to be available to real developers, I developed a software environment to support the construction and use of ASTRAL formal specifications [35]. This software development environment can be used to specify, design, and verify complex real-time software systems.

We also implemented a model checker for ASTRAL, which was integrated into the software development environment. It allows a user to test formal specifications written in the ASTRAL formal language [7]. This enables system developers to find problems early in the software development life cycle before they are too costly to fix.

Result: Technique, Tool

Lesson learned: *Build on previous projects.*

2.9. Intrusion Detection

2.9.1 State Transition Analysis technique

In the early 1990s, I and my students developed a new approach to modeling computer penetrations, called the State Transition Analysis Technique (STAT) [40]. The first implementation of an intrusion detection system based on the STAT approach was built for the SunOS 4.1.1 operating system [21]. This system was later ported to Solaris and Windows NT. In the late nineties we extended the State Transition Analysis Technique so that it could detect network-based attacks [43, 44]. This system is called NetSTAT.

After developing a number of intrusion detection systems for various platforms and domains, we decided to concentrate on developing a framework for constructing a family of intrusion detection systems [47]. The framework is comprised of an extensible state/transition-based attack description language (STATL) that allows one to describe computer penetrations as sequences of actions that an attacker performs to compromise a computer system, a core model that implements the STATL semantics, and a communication and control infrastructure for controlling a network of sensors.

A STATL description of an attack scenario can be used by an intrusion detection system to analyze a stream of events and detect possible ongoing intrusions. Since intrusion detection is performed in different domains (i.e., the network or the hosts) and in different operating environments (e.g., Linux, Solaris, or Windows NT) it is important to have an extensible language that can be easily tailored to different target environments. STATL defines domain-independent features of attack scenarios and provides constructs for extending the language to describe attacks in particular domains and environments. The STATL language has been successfully used in describing network-based, host-based and application-based attacks, and it has been tailored to very different environments, e.g., Sun Microsystems' Solaris and Microsoft's Windows NT. The details of the STATL syntax and semantics can be found in [13].

An implementation of the runtime support for the STATL language has also been developed, and a family of intrusion detection systems based on STATL has been implemented [46, 42]. These systems vary from a network-based intrusion detection system, to host-based and application-based systems, to alert correlators. Tools in the family have been used in a number of DARPA-sponsored intrusion detection evaluation efforts, and they have always delivered impressive results. These results demonstrated that by using the STAT framework it is possible to develop intrusion detection systems with reduced development effort, with respect to *ad hoc* approaches. In addition, the framework-based approach is advantageous in terms of the increased reuse that results from using a component-based architecture [47]. The STAT core code, as well as the extensions for various host-based and network-based systems, is available at the Computer Security Group web page (www.cs.ucsb.edu/∼seclab/). This code base is approximately 300,000 lines of code. By making these systems widely available, other researchers and practitioners can easily experiment and gain first-hand experience with their capabilities.

2.9.2 MetaSTAT

Intrusion detection relies on the information provided by a number of sensors deployed throughout the monitored network infrastructure. Sensors provide information at different abstraction levels and with different semantics. In addition, sensors range from lightweight probes and simple log parsers to complex software artifacts that perform sophisticated analysis. Managing a configuration of heterogeneous sensors can be a very time-consuming task. Management tasks include planning, deployment, initial configuration, and run-time modifications. To support the fine-grained configuration of STAT-based sensors, we developed an infrastructure called MetaSTAT, which provides a network security officer with the capability to perform sensor configuration remotely [25]. MetaSTAT also supports the explicit modeling of the dependencies among the modules composing a sensor so that it is possible to automatically

identify the steps that are necessary to perform a reconfiguration of the deployed sensing infrastructure. This provides a level of dynamic configurability that is superior to any other existing system.

The result of applying the STAT/MetaSTAT approach is a "web of sensors", composed of distributed intrusion detection systems integrated by means of a communication and control infrastructure [45]. The task of a web of sensors is to provide coordinated security monitoring of very diverse event streams in a protected network. Multiple webs of sensors can be organized either hierarchically or in a peer-to-peer fashion to achieve scalability and to be able to exert control over a large-scale infrastructure from a single control location [26].

2.9.3 High-Speed Networks

As networks become faster there is an emerging need for security analysis techniques that can keep up with the increased network throughput. Existing network-based intrusion detection sensors can barely keep up with bandwidths of a few hundred Mbps. Analysis tools that can deal with higher throughput are unable to maintain state between different steps of an attack or they are limited to the analysis of packet headers. The Computer Security Group at UCSB also addressed the problems of performing intrusion detection on high-speed networks. We used a partitioning approach to network security analysis that supports in-depth, stateful intrusion detection on high-speed links. The approach is centered around a slicing mechanism that divides the overall network traffic into subsets of manageable size. The traffic partitioning is done so that a single slice contains all the evidence necessary to detect a specific attack, making sensor-to-sensor interactions unnecessary. The details of the approach and an experimental evaluation of its effectiveness can be found in [36].

Result: Technique, Tools

Lesson learned: *You should work on problems that are interesting to you, whether they are funded or not. If you do a good job, you will eventually be funded.*

After the second or third "one of" consider using a software family approach.

2.10. Red Teaming

As more applications are distributed and network access becomes the norm, new security problems arise. During the Christmas break of 1996, I had a visitor from Italy who was coming for 6 weeks. So that we could get the most out of his visit, I asked what areas he was most interested in. One of the areas was web applications. Since I knew very little about web applications at the time, I suggested that we look at what had been done in the area of web security.

During this six week period we discovered several fatal flaws in the then current web browser technologies [39]. One of these flaws allowed complete access to all data entered by an unsuspecting browser user. This was an example of spyware, before the term had been coined. We discovered this flaw two days after Netscape released the browser, and Netscape came up with a new release within two weeks of being notified of the flaw. Netscape named this flaw the "Santa Barbara Privacy Bug."

Motivated by the flaws that we discovered in web applications, one of my PhD students, André dos Santos, and I developed an approach, based on smart cards, that allows one to interact with otherwise unsafe applications in a secure way. The general approach is called Safe Areas of Computation (SAC), and it was first presented at ACSAC99 [10].

The initial WWW work also resulted in getting to evaluate the security of an online banking application for a large international bank. Starting from a single legitimate account and unprivileged Internet access to the bank's site we demonstrated how it was possible to compromise the security of many accounts and transfer funds to our own account [9]. We also showed how by using the SAC approach most of the compromises with the bank application could be avoided.

My most recent red team experience was this past summer when the California Secretary of State asked our research group to be part of the Top-to-Bottom Review of California's electronic voting systems. The Computer Security Group knew very little about voting systems, but we knew a lot about red teaming.

The group performed a series of security tests of both the hardware and the software that are part of the Sequoia electronic voting system to identify possible security problems that could lead to a compromise. In this case, a "compromise" was defined as "tampering or an error that could cause incorrect recording, tabulation, tallying or reporting of votes or that could alter critical election data such as election definition or system audit data."

The team was able to expose a number of serious security issues. We were able to bypass both the physical and the software security protections of the Sequoia system, and we demonstrated how these vulnerabilities could be exploited by a determined attacker to modify (or invalidate) the results of an election.

The complete red team report, as well as the reports for the Diebold and Hart systems, which were evaluated by other teams, is available online at
http://www.sos.ca.gov/elections/elections_vsr.htm.

Result: Technique, Product

Lesson learned: *By working on real systems, you can have a large impact.*

Be a good citizen.

Don't be afraid to move into new areas.
Don't overlook the obvious.
Being the attacker is a lot of fun.

3. Future Security Challenges

In the early days of computing, when standalone systems were used by one user at a time, computer security consisted primarily of physical security. That is, the computer and its peripherals were locked in a secure area with a guard at the door that checked each user's identification before allowing them to enter the room. Unfortunately, as computers and computer applications got more complex and thousands, and even millions, of systems were connected to one another, protecting these networked systems became a daunting task.

Most security experts (ask the person sitting next to you) already agree that, given all of the desirable user features, such as network connectivity and user friendliness, the goal of having a system that is "completely secure" (whatever that means) will never be achieved. But cheer up. Things are just going to get worse. I think two of the biggest problems that we security researchers have to face our *privacy* and *ubiquitous computing*.

The incredible growth in the use of the Internet has resulted in an increase in the number of privacy vulnerabilities. As users are given online access to their bank accounts, health records, and shopping portals, the demand for more user friendly interfaces has increased. To make these systems more user friendly, personal information, such as shopping preferences, and even credit card and social security numbers, are collected and stored on the systems. The problem is that the same information that is used to make access more efficient and user friendly is also accessible to the attackers. As a result, the number of attacks aimed at accessing personal data has also increased dramatically. These attacks range from the release of embarrassing personal information to identity theft and personal financial losses. Controlling the distribution and use of personal information is already a big problem, and it is only going to get worse. The controls have to allow each person to decide how they want their data to be handled and distributed.

Now let's consider ubiquitous or pervasive computing. The integration of computation into the environment, rather than having computers as distinct objects, opens up a whole new class of security and privacy problems. The selling point of the promoters of ubiquitous computing is that embedding computation into the environment will enable people to move around and interact with computers more naturally than they currently do. Imagine trying to be security conscious when you do not even know that you are currently interacting with one (or many) of these embedded systems.

One of the goals of the pervasive computing community is to enable devices to sense changes in the environment and to automatically adapt and respond based on these changes and based on the user's needs and preferences. Do you trust the systems you are interacting with (possibly without being aware of the interaction) to know what your preferences are? Also, who controls the dissemination of your preference information? How can you assure the privacy of your personal information (e.g., preferences) if you are not aware of who or what holds this information? There is no doubt that the increased complexity of these pervasive systems will make the task of assuring the security of these systems orders of magnitude more than what we currently need for today's systems.

So what are we going to do? As I said in the previous subsection, *Don't be afraid to move into new areas.* I view these problems as fun new areas to start working in. Want to join me?

4. Conclusions

I would like to reiterate that it is important to ground your research in practice. The goal of every research project should be that it produces something that others can use. To test the practicality of your research, ask the question "Would a practitioner find my technique, tool, or product useful?".

In this paper I have summarized some of the many research projects that I have worked on and pointed out what I thought were some lessons learned. Now, after selling you on the idea of grounding your research in practice, I feel that I need to come clean. That is, *not all* research that is useful ends up with an identifiable technique, tool, or product. This is particularly true of some foundational research, but it is also true of more practical projects, such as the Verification Assessment Study reported in section2.5. Furthermore, it is often the case that the use is not discovered until years later. This does not mean, however, that you should not ask the question of practitioner relevance when choosing a research direction.

I would like to make another point, particularly for the younger faculty and students in the audience. Do not let the available research dollars drive your choice of research directions. Good research is not always funded. When I first started the STAT approach, it was not funded. In fact, it was unfunded for about six years. Also, don't confuse unfunded and not fundable; I eventually received more than seven million dollars in funding for this work.

I also need to mention that the term "product" as used in this paper does not mean something that is ready to be sold on the open market. One of the research projects that I did not discuss in this paper is our work on web application firewalls. Over the past year or two we have been commercializing this research through a startup. A lesson

that I have learned from this exercise is that productizing a research tool is a long and difficult process. It is, however, quite a rewarding learning experience and a lot of fun.

Finally, I would like to thank the many colleagues that I have worked with on the projects presented in this paper and on other projects that I did not include. In particular, I would like to acknowledge all of the current and past members of the Computer Security Group (formerly known as the Reliable Software Group) at UCSB.

References

[1] B. Auernheimer and R. A. Kemmerer. Aslan user's manual. Technical Report TRCS84-10, Department of Computer Science, University of California, Santa Barbara, CA, 1985.

[2] B. Auernheimer and R. A. Kemmerer. Procedural and non-procedural semantics of the aslan formal specification language. In *Proceedings of Nineteenth Annual Hawaii International Conference on System Sciences*, January 1986.

[3] A. Coen-Porisini, C. Ghezzi, and R. A. Kemmerer. Specification of realtime systems using astral. *IEEE Trans. Software Eng.*, 23(9):572–598, 1997.

[4] A. Coen-Porisini and R. A. Kemmerer. The composability of astral realtime specifications. In *ISSTA*, pages 128–138, 1993.

[5] A. Coen-Porisini, R. A. Kemmerer, and D. Mandrioli. A formal framework for astral intralevel proof obligations. *IEEE Trans. Software Eng.*, 20(8):548–561, 1994.

[6] A. Coen-Porisini, R. A. Kemmerer, and D. Mandrioli. A formal framework for astral inter-level proof obligations. In W. Schäfer and P. Botella, editors, *ESEC*, volume 989 of *Lecture Notes in Computer Science*, pages 90–108. Springer, 1995.

[7] Z. Dang and R. A. Kemmerer. Using the astral model checker to analyze mobile ip. In *ICSE*, pages 132–142, 1999.

[8] L. Dillon, R. A. Kemmerer, and L. Harrison. An experience with two symbolic execution-based approaches to formal verification of ada tasking programs. In *Proceedings of Software Testing, Verification, and Analysis*, pages 114–122, Los Alamitos, CA, July 1988. IEEE Computer Society Press.

[9] A. dos Santos, G. Vigna, and R. Kemmerer. Security Testing of an Online Banking Service. In A. Ghosh, editor, *E-Commerce Security and Privacy*, Advances in Information Security, pages 3–15. Kluwer Academic Publishers, 2001.

[10] A. L. M. dos Santos and R. A. Kemmerer. Safe areas of computation for secure computing with insecure applications. In *ACSAC '99: Proceedings of the 15th Annual Computer Security Applications Conference*, page 35, Washington, DC, 1999. IEEE Computer Society.

[11] J. Douglas and R. A. Kemmerer. Aslantest: a symbolic execution tool for testing aslan formal specifications. In *ISSTA '94: Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, pages 15–27, New York, NY, USA, 1994. ACM Press.

[12] S. T. Eckmann and R. A. Kemmerer. Inatest: an interactive environment for testing formal specifications. *SIGSOFT Softw. Eng. Notes*, 10(4):17–18, 1985.

[13] S. T. Eckmann, G. Vigna, and R. A. Kemmerer. Statl: An attack language for state-based intrusion detection. *Journal of Computer Security*, 10(1/2):71–104, 2002.

[14] R. W. Erickson and D. R. Musser. The affirm theorem prover: Proof forests and management of large proofs. In *Proceedings of the 5th Conference on Automated Deduction*, pages 220–231, London, UK, 1980. Springer-Verlag.

[15] R. W. Floyd. Assigning meaning to programs. In *Proceedings of Symposia in Applied Mathematics*, 1967.

[16] J. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the 1982 Symposium on Security and Privacy*, pages 11–20, Los Alamitos, CA, 1982. IEEE.

[17] G. Grossman. A practical executive for secure communications. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, pages 144–155. IEEE, 1982.

[18] J. Haigh, R. Kemmerer, J. McHugh, and W. Young. An experience using two covert channel analysis techniques on a real system design. *IEEE Transactions on Software Engineering*, 13(2), February 1987.

[19] L. Harrison and R. A. Kemmerer. An interleaving symbolic execution approach for the formal verification of ada programs with tasking. In *Proceedings of Third International IEEE Conference on Ada Applications and Environments*, pages 15–26, Los Alamitos, CA, USA, May 1988. IEEE Computer Society Press.

[20] T. Hinke, J. Althouse, and R. A. Kemmerer. Sdc secure release terminal project. In *Proceedings of the 1983 IEEE Symposium on Security and Privacy*, Los Alamitos, CA, 1983. IEEE Computer Society.

[21] K. Ilgun, R. A. Kemmerer, and P. A. Porras. State transition analysis: A rule-based intrusion detection approach. *IEEE Trans. Softw. Eng.*, 21(3):181–199, 1995.

[22] D. Kalisch and R. Montague. *Logic techniques of formal reasoning*. Harcourt, Brace, and World, Inc., New York, NY, 1964.

[23] R. Kemmerer and P. Porras. Covert flow trees: A visual approach to analyzing covert storage channels. *IEEE Transactions on Software Engineering*, SE-17(11):1166–1185, November 1991.

[24] R. Kemmerer and T. Taylor. A modular covert channel analysis methodology for trusted DG/UX. In *Proceedings of the Twelfth Annual Computer Security Applications Conference*, pages 224–235, December 1996.

[25] R. Kemmerer and G. Vigna. Hi-DRA: Intrusion Detection for Internet Security. *IEEE Proceedings*, 93(10):1848–1857, October 2005.

[26] R. Kemmerer and G. Vigna. Hi-DRA: Intrusion Detection for Internet Security. *IEEE Proceedings*, 93(10):1848–1857, October 2005.

[27] R. A. Kemmerer. Shared resource matrix methodology: an approach to identifying storage and timing channels. *ACM Trans. Comput. Syst.*, 1(3):256–277, 1983.

[28] R. A. Kemmerer. Testing formal specifications to detect design errors. *IEEE Trans. Software Eng.*, 11(1):32–43, 1985.

[29] R. A. Kemmerer. Verification assessment study final report. Technical Report C3-CR01-86, National Computer Security Center, Ft. Meade, MD, 1986. 5 Volumes (Overview, Gypsy, Affirm, FDM, and EHDM). US distribution only.

[30] R. A. Kemmerer. Using formal verification techniques to analyze encryption protocols. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, Washington, DC, USA, 1987. IEEE Computer Society.

[31] R. A. Kemmerer. Analyzing encryption protocols using formal verification techniques. *IEEE Journal on Selected Areas in Communications*, 7(4):448–457, 1989.

[32] R. A. Kemmerer and S. T. Eckmann. Unisex: A unix-based symbolic executor for pascal. *Software: Practice and Experience*, 15(3):439–458, 1985.

[33] R. A. Kemmerer, C. Meadows, and J. Millen. Three systems for cryptographic protocol analysis. *Journal of Cryptography*, 7(2):79–130, 1994.

[34] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[35] P. Z. Kolano, Z. Dang, and R. A. Kemmerer. The design and analysis of real-time systems using the astral software development environment. *Ann. Software Eng.*, 7:177–210, 1999.

[36] C. Krügel, F. Valeur, G. Vigna, and R. A. Kemmerer. Stateful intrusion detection for high-speed networks. In *IEEE Symposium on Security and Privacy*, pages 285–, 2002.

[37] R. Locasso, J. Scheid, V. Schorre, and P. Eggert. The ina jo specification language reference manual. In *SDC document TM-6889/000/01*, 1980.

[38] C. Meyer and S. Matyas. *Cryptography*. Wiley, New York, NY, 1980.

[39] F. D. Paoli, A. L. M. dos Santos, and R. A. Kemmerer. Web browsers and security. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 235–256. Springer, 1998.

[40] P. Porras and R. Kemmerer. Penetration state transition analysis: A rule-based intrusion detection approach. In *ACSAC '92: Proceedings of the 8th Annual Computer Security Applications Conference*, pages 220–229, Washington, DC, 1992. IEEE Computer Society.

[41] M. Shaw, W. A. Wulf, and R. L. London. Abstraction and verification in alphard: defining and specifying iteration and generators. *Commun. ACM*, 20(8):553–564, 1977.

[42] G. Vigna, S. Gwalani, K. Srinivasan, E. M. Belding-Royer, and R. A. Kemmerer. An intrusion detection tool for aodv-based ad hoc wireless networks. In *ACSAC*, pages 16–27. IEEE Computer Society, 2004.

[43] G. Vigna and R. A. Kemmerer. Netstat: A network-based intrusion detection approach. In *ACSAC*. IEEE Computer Society, 1998.

[44] G. Vigna and R. A. Kemmerer. Netstat: A network-based intrusion detection system. *Journal of Computer Security*, 7(1), 1999.

[45] G. Vigna, R. A. Kemmerer, and P. Blix. Designing a web of highly-configurable intrusion detection sensors. In W. Lee, L. Mé, and A. Wespi, editors, *Recent Advances in Intrusion Detection*, volume 2212 of *Lecture Notes in Computer Science*, pages 69–84. Springer, 2001.

[46] G. Vigna, W. Robertson, V. Kher, and R. Kemmerer. A Stateful Intrusion Detection System for World-Wide Web Servers. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC 2003)*, pages 34–43, Las Vegas, NV, December 2003.

[47] G. Vigna, F. Valeur, and R. A. Kemmerer. Designing and implementing a family of intrusion detection systems. In *ESEC / SIGSOFT FSE*, pages 88–97, 2003.

[48] B. J. Walker, R. A. Kemmerer, and G. J. Popek. Specification and verification of the ucla unix security kernel. *Commun. ACM*, 23(2):118–131, 1980.

[49] W. A. Wulf, R. L. London, and M. Shaw. An introduction to the construction and verification of alphard programs. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, page 390, Los Alamitos, CA, 1976. IEEE Computer Society Press.