Towards Simple and Scalable Analysis of Secure Systems

Jason Franklin, Deepak Garg, Dilsun Kaynar, Anupam Datta
Carnegie Mellon University

1 Introduction

Contemporary secure systems are complex and designed to provide subtle security properties in the face of attack. Examples of such systems include virtual machine monitors, security kernels, web browsers, and secure co-processor-based systems such as those utilizing the Trusted Computing Group's Trusted Platform Module (TPM) [13]. In many cases, it is unclear what security guarantees are offered by such systems and against what class of adversaries. The common practice of enumerating known attacks coupled with informal security arguments is clearly unsatisfactory.

In this paper, we provide an overview of a program that we have initiated to develop *simple*, *sound*, and *scalable* techniques for proving the security of networked system designs. We report on our preliminary experiences with developing and applying the *Logic of Secure Systems (LS^2)* and outline our vision for this research program.

Overcoming the complexity of systems remains a central challenge in system modeling and analysis. Despite substantial progress, the goal of proving deep properties of system implementations remains a major challenge. As an alternative, we make the cause for the use of aggressive abstraction: modeling and analysis of system designs at a high-level where in-depth analysis can identify flawed designs prior to implementation.

We designed LS^2 to simplify the process of abstracting away unnecessary details that complicate modeling and analysis and to reveal high-level abstractions of security-relevant systems components which we term *security skeletons*. The security skeleton of a system is a sparse specification and includes just the code responsible for security-relevant operations. LS^2 further reduces modeling effort by natively including common systems primitives as language constructs such as shared memory, memory protection, machine resets, cryptographic operations, network communication, control flow, and dynamic loading and executing of code.

To analyze the security of a system, one first needs to specify the property or properties of interest. Unfortunately, there is a paucity of general definitions of security for systems. This is in part due to the diversity of systems, but also due to the fact that most systems analysis is performed by enumerating specific attacks rather than proving a system secure against a general class of attackers specified by their capabilities. One goal of LS^2 is to enable the exploration of definitions of security for a wide variety of systems ranging from web browsers to virtual machine monitors and hypervisors.

To be secure, a system must satisfy a security definition *in the presence of an adversary*. The inclusion of an adversary into an analysis expands the possible states of the system beyond those of the system alone. As a result, analysis without an adversary or analysis of a system's resistance to specific attacks may result in overlooking possible vulnerable system states. One goal of LS^2 is to encourage exploration and standardization of systems adversaries. In addition to including a network adversary adopted from protocol analysis, LS^2 includes a local attacker modeled as a malicious local thread. The adversaries are specified by their capabilities, not the specific attacks they can perform. As a result, a security proof guarantees a system is secure against all attacks the adversaries can perform. To support scalable reasoning and reduce the effort required to prove designs secure, a sound proof system was developed that allows reasoning about system security without explicitly considering the actions of adversaries.

We have applied LS^2 to analyze trusted computing systems utilizing both TPMs and next generation hardware-support for the late launch of a security kernel [12, 1, 8]. Our analysis clarifies ambiguous details of the specification, explicitly states the requirements underlying the security of the system, and discovers previously unknown security vulnerabilities in a class of deployed trusted computing systems. After modifying the system specifications to fix the vulnerabilities, we prove that the augmented system satisfies a natural security property.

We are in the process of extending LS^2 to further simplify modeling and analysis and increase expressiveness. We are also adding support for tunable adversaries specified by the set of interfaces they are allowed to access. After completing these extensions, we intend to model and analyze the security of systems including web browsers, hypervisors, and virtual machine monitors, in the process developing new adversary models and formal definitions of security.

2 Modeling Secure Systems

In LS^2 systems are modeled in a programming language. A secure system is specified as a set of programs in this language. Each program consists of a number of actions that are executed in a straight line. For example, a trusted computing system contains two programs, one to be executed by the untrusted platform and the other by the remote verifier. A single executing program is called a *thread*. The programming language is designed to be *expressive* enough to model practical secure systems while still maintaining a sufficiently high level of abstraction to enable *simple reasoning*.

2.1 Language Constructs

Following its predecessor, PCL [4], the language includes process calculi and functional constructs for modeling cryptographic operations, straightline code, and network communication among concurrent processes. We introduce new constructs for modeling machines and shared memory, a simple form of access control on memory, machine resets, and dynamically loading and executing unknown (and potentially untrusted) code. The primitives for reading and writing to memory are inspired by the treatment of memory cells in impure functional languages like Standard ML [10]. We model memory protection, a fundamental building block for secure systems [11], by allowing programs to acquire exclusive-write locks on memory cells. The treatment of dynamically loading and executing unknown and untrusted code is novel to this work. The ability to model the execution of unknown and untrusted code is integral to faithfully modeling security threats against web browsers including malicious javascript and plugins, program isolation in virtual machines, and code attestation protocols in trusted computing.

While these constructs are the common denominator for many secure systems, they are by no means sufficient to model all systems of interest. The language, however, is *extensible* in a modular fashion, as we have illustrated by extending the core language with a trusted computing subsystem. At a high level, each system component can be viewed as exposing an *interface*. For example, the interface for memory includes read, write and reset operations. Adding a new component to the system involves adding operations in the programming language corresponding to the interface exposed by it. For example, Platform Configuration Registers (PCRs) in the TPM can be modeled as a special form of memory that can be accessed via read, reset and a new extend operation. Some extensions can have a more global effect on the language semantics. For example, adding the machine reset operation to the language affects both how state of local memory and TPM PCRs may be updated. We describe below the core language constructs.

Straightline code and cryptography. Straight line code execution is modeled using sequences of actions that perform standard operations like signing and signature verification, encryption and decryption (both symmetric and asymmetric), nonce generation, hashing, value matching, pairing and projection. Each action returns a value, which may be given a name to refer to the value in subsequent actions. Our model of straightline code execution is thus *functional*. This design choice simplifies reasoning significantly.

Network primitives. Threads can communicate using actions to send and receive values over the network. Network communication is untargeted, i.e., any thread may intercept and read any message (dually, a received message could have been sent by any thread). Information being sent over the network may be protected using cryptography, if needed. The treatment of network communication and cryptography follows PCL. The language constructs we present next are new to this work.

Machines and shared memory. Threads can also share data through shared memory. The programming model contains machines explicitly. Each machine contains a number of memory cells that are shared by all threads running on the machine. These cells may be classified into RAM, persistent store (hard disk), or other special purpose cells (such as PCRs). Depending on the type of cell, its behavior may be slightly different. For example, RAM cells are set to a fixed value when a machine resets, whereas persistent store is not affected by resets. Despite these differences, the prominent characteristics of all cells are that they can be *read* and *written* through actions provided in the programming language, and that they are *shared* by all threads on the machine. Consequently, any thread (including an adversarial thread) has the potential to read or modify any cell.

Access control on memory. Shared memory, by its very nature, cannot be used in secure programs unless some access control mechanism enforces the integrity and confidentiality of data written to it. Access control varies by type of memory and application (e.g., memory segmentation, page table read-only bits, access control lists in file systems, etc). Our programming model provides an abstract form of access control through *locks*. Any running thread may obtain an exclusive write lock on any previously unlocked memory cell by executing a single action provided for this purpose. The semantics of the programming language guarantee that while the lock is held by the thread, no other thread will be able to write the cell. The thread may later relinquish the lock it holds by executing another action. Locking in this manner may be used to enforce *integrity* of contents of memory. In a similar manner, one may add read locks that provide *confidentiality* of memory contents. Our abstract locks, although simple, are a faithful model of the memory protection that hardware usually provides. They can be used to build other forms of access control such as page tables and access control lists if needed.

Machine resets. The language allows a machine to be spontaneously reset. When a machine reset occurs in our model, all running threads on it are killed, all its RAM cells are set to a fixed value, and a single new thread is created to reboot the machine. This new thread executes a fixed booting program. We model the reset operation since it has significant security implications for secure systems [3]. In the context of trusted computing, e.g., the fact that a TPM's PCRs are set to a fixed value is critical in reasoning about the security properties of attestation protocols. In addition, it has been shown that adversaries can launch realistic attacks against trusted computing systems using machine resets [5].

Untrusted code execution. The last salient feature of our programming model is an action that dynamically branches to code. The code branched to is represented by an arbitrary value, that may reside in memory or on disk, or even be received over the network. Note that this code could have come from an adversary. Execution of untrusted code is necessary to model several systems of interest, e.g., trusted computing systems and web browsers.

Operational semantics. The abstract runtime environment of the language is called a *configuration*: It contains all the executing threads, the state of memory on all machines, and the state of memory locks held by threads. The operational semantics of the language captures how systems execute to produce traces. It is defined using process calculus-style *reduction rules* that specify how a configuration may transition to another.

Example Model. Figure 1 shows an example security skeleton of a trusted computing system that utilizes a TPM and hardware support for late launch. The protocol includes four agents executing a number of processes including: (1) $OS(m)$, executed by the machine itself (called $\hat{m}$), that receives a nonce from the remote verifier, and performs a late launch. (2) $LL(m)$, executed by the hardware platform, that acquires exclusive write locks on the PCRs, performs a dynamic reset of PCR $m.dpcr.k$, reads the binary of the program P from the secure loader block (SLB), and measures or in trusted computing parlance, extends, then calls P, (3) $P(m)$ that measures the nonce, evaluates the function f on input 0 (the function f and its input may be replaced by some other function depending on application), and extends a distinguished string EOL into $m.dpcr.k$ to signify the end of the late launch session. (4) $TPM(m)$, executed by the TPM of m, that signs the dynamic PCR $m.dpcr.k$, and sends it to the verifier. (5) $Verifier(m)$, executed by a remote verifier, that generates and sends a nonce, receives signed integrity measurements, verifies the signature, and checks that the measurements match the expected sequence $(dinit, P(m), n, EOL)$.

$$
\begin{array}{lcl}
OS(m) & \equiv & n' = \texttt{receive}\ ;\\
 & & \texttt{write}\ m.nonce, n';\\
 & & \texttt{late\_launch}\\[1em]
LL(m) & \equiv & P = \texttt{read}\ m.SLB;\\
 & & \texttt{extend}\ m.dpcr.k, P;\\
 & & \texttt{call}\ P\\[1em]
P(m) & \equiv & n'' = \texttt{read}\ m.nonce;\\
 & & \texttt{extend}\ m.dpcr.k, n'';\\
 & & \texttt{eval}\ f, 0;\\
 & & \texttt{extend}\ m.dpcr.k, EOL\\[1em]
TPM(m) & \equiv & w = \texttt{read}\ m.dpcr.k;\\
 & & r = \texttt{sign}\ (dPCR(k), w), AIK^{-1}(m);\\
 & & \texttt{send}\ r\\[1em]
Verifier(m) & \equiv & n = \texttt{new}\ ;\\
 & & \texttt{send}\ n;\\
 & & sig = \texttt{receive}\ ;\\
 & & v = \texttt{verify}\ sig, AIK(m);\\
 & & \texttt{match}\ v, (dPCR(k),\\
 & & \quad seq(dinit, P(m), n, EOL))
\end{array}
$$

Figure 1: Security Skeleton for Trusted Computing System

2.2 Adversary Model

We formally model adversaries as extra threads executing concurrently with protocol participants. Such an adversary may contain any number of threads, on any machines, and may execute any program expressible in our programming model. However, the adversary cannot perform operations that are not permitted by the language semantics. For example, the adversary can neither write to memory locked by another thread, nor can it break cryptography.

Interfaces to system components also provide a useful conceptual view of the *adversary*. Since the capabilities of the adversary are constrained by the system interface, we refer to her as a CSI-ADVERSARY. For example, the adversary can write to unprotected memory cells, but can only update PCR's through the extend operation in its interface. Formally, the adversary may execute any program expressible in our programming model, i.e. the adversary can perform symbolic cryptographic operations, intercept and inject messages that it can create into the network, read and write memory cells that are not explicitly locked by another thread, and reset machines. Because of these capabilities, the adversary can launch a broad range of attacks on the network and the local machines including replay attacks, modifying and injecting malicious code on local machines, and exploiting race conditions.

3 Analysis

Despite the complexity inherent in reasoning about security in the face of adversaries that are generically specified only by their capabilities, reasoning principles in LS^2 are fairly easy to use. Technically, this is accomplished through a combination of an intuitive and simple *proof system* that is used to prove security properties, and a system-independent *soundness theorem* that connects the proof system to execution of programs and, more significantly, captures the complexity of reasoning about actions of adversaries. The latter theorem, although non-trivial, has been established once and for all. As a consequence of the theorem, proofs of security properties proceed by induction on programs of known system components only, without ever having to consider adversarial actions. This simplifies reasoning significantly, and makes LS^2 amenable to use in practical systems.

Formally, security of a system is specified as safety properties of its components, which are further expressed as invariants of the programs of the components. These invariants are similar to program invariants in Hoare logic [7]. A proof of a security property for a component consists of an induction on the program of the component using fixed rules and axioms (that constitute the proof system). These rules and axioms are designed to be expressive enough to prove most properties of interest, yet simple to understand. For example, a literal English translation of one of the axioms for reasoning about values stored in memory is the following: "If only thread T can write to a memory cell l during a time interval i and T does not write to l during i, then l must contain the same value throughout i." Similarly, an axiom for reasoning about memory protection reads, "If thread T has an exclusive-write lock on memory cell l at the beginning of time interval i, and T does not relinquish its lock during i, then l must hold the lock throughout i."

Like these two axioms, all other axioms and rules in LS^2 are fairly obvious. The technical difficulty lies in proving that these axioms are sound, i.e, that the properties they state actually hold when programs execute. For example, in the case of the first axiom above, this amounts to showing that no matter what the adversary and other concurrently executing programs try to do, the value in cell l will not change during i. This requires an exhaustive induction on the possible programs of the adversary, which is both tedious and non-trivial. However, as mentioned earlier, this complexity is dealt with once in the soundness theorem; proofs of security of systems simply *use* the axioms and rules without paying any heed to their correctness and are, therefore, straightforward.

Based on our experience with LS^2, we believe that its use requires an initial learning phase, but little subsequent effort for application to new systems. As an example, one member of our group who had no prior familiarity with program logics was able to use LS^2 to prove security properties of several core components of TPMs with only a moderate effort after an initial intense phase of learning the formal syntax and proof system. We believe that this will be the case for other system designers as well. Whereas the difficulty with initial learning is inherent in logical syntax and formal proofs, ease of subsequent use is a consequence of careful design of the reasoning principles used in LS^2.

Example Analysis. We formally define an integrity property of the system in Figure 1. We summarize the system security property as follows: if the verifier is not the TPM, the TPM does not leak its signing key, and the TPM executes only the processes $TPM(m)$ and $TPM_{SRTM}(m)$[1], then after executing its code successfully, the remote verifier is guaranteed that J performed a single late launch on machine m at some time t_L, J called $P(m)$ only once at t_C, J evaluated f once at t_E (and this happened after the verifier generated the nonce), J extended EOL into $m.dpcr.k$ at time t_X, and $m.d.pcr.k$ was locked for the thread J from t_L to t_X. We formalize this security property called J_{TC} below.

$$
\begin{array}{ll}
[Verifier(m)]_V^{t_b,t_e} & \exists J,t_X,t_E,t_N,t_L,t_C,n. \\
& \wedge\ (t_L < t_C < t_E < t_X < t_e) \\
& \wedge\ (t_b < t_N < t_E) \\
& \wedge\ (\mathsf{New}(V,n)\ @\ t_N) \\
& \wedge\ (\mathsf{LateLaunch}(m,J)\ @\ t_L) \\
& \wedge\ (\neg\mathsf{LateLaunch}(m)\ \mathsf{on}\ (t_L,t_X]) \\
& \wedge\ (\neg\mathsf{Reset}(m)\ \mathsf{on}\ (t_L,t_X]) \\
& \wedge\ (\mathsf{Call}(J,P(m))\ @\ t_C) \\
& \wedge\ (\neg\mathsf{Call}(J)\ \mathsf{on}\ (t_L,t_C)) \\
& \wedge\ (\mathsf{Eval}(J,f)\ @\ t_E) \\
& \wedge\ (\mathsf{Extend}(J,m.dpcr.k,EOL)\ @\ t_X) \\
& \wedge\ (\neg\mathsf{Eval}(J,f)\ \mathsf{on}\ (t_C,t_E)) \\
& \wedge\ (\neg\mathsf{Eval}(J,f)\ \mathsf{on}\ (t_E,t_X)) \\
& \wedge\ (\mathsf{IsLocked}(m.dpcr.k,J) on (t_L,t_X])
\end{array}
$$

The proof of the above property consists of only around fifteen steps and is concise enough to fit into five pages of text. The proof itself required less than one day of effort to establish. While performing the proof, we determined that recently introduced hardware support for late launch actually adversely affects the security of previous generation trusted computing systems! In particular, an attack is possible that utilizes late launch hardware to exploit load-time attestation protocols that measure software starting at system boot. By performing a late launch after the operating system has been measured into a PCR, a malicious program can modify the suspended OS binary without detection. The attack enables an adversary to report false system integrity measurements that are not tied to the actual state of the platform. This vulnerability could be countered if the suspended OS were able to maintain memory protections over the state previously measured into the static PCRs, unfortunately current generation hardware does not provide this option.

[1] TPM_{SRTM} is not shown.

4 Related Work

LS^2 shares a number of features with PCL [4] and therefore with other logics of programs [7, 6, 2]. One central difference from PCL is that LS^2 considers shared memory systems in addition to network communication. Although concurrent separation logic [2] also focuses on shared-memory programs with mutable state, a key difference is that it does not consider network communication. Furthermore, concurrent separation logic and other approaches for verifying concurrent systems [9] typically do not consider an adversary model. An adversary could be encoded as a regular program in these approaches, but then proving invariants would involve an induction over the steps of the honest parties programs and the attacker. On the other hand, in LS^2 (as in PCL), invariants are established only by induction over the steps of honest parties programs, thereby considerably simplifying the analysis.

5 Ongoing Work

Although we are satisfied with the overall approach of LS^2, we have noticed a number of limitations of its programming model that we are in the process of improving.

Interfaces. First, LS^2 currently cannot be used to reason about system-specific interfaces. Instead reasoning must use LS^2's in-built primitives. For example, LS^2 provides a 'write' primitive that models writing a cell on disk directly (without any understanding of the directory structure). However, in modeling a real file system, it may be more useful to define and reason about a POSIX style "write" system call that maintains invariants of the inode structure. Currently, this cannot be done in LS^2. More significantly, the adversary's capabilities are defined exactly by the primitives available in the programming model, not the system interfaces that are accessible to the adversary. Whereas this is appropriate in some cases, in others it makes the adversary stronger than necessary in practice. In ongoing work, we are extending LS^2 with methods for describing system interfaces as combinations of specifications and programs, and for reasoning about security with adversaries that are confined to a stipulated but variable set of interfaces.

Access Control. The second major improvement that we would like to make is the addition of a flexible model for access control. Currently, the only primitive for access control in LS^2 is an exclusive-write lock on a RAM or disk cell that prevents all but the owning process from writing the cell. Although fundamental to building more sophisticated access control in systems, this is a rather low-level specification of protection, and is limiting, for example, when we want to reason about cells shared by multiple processes. In future, we would like to enrich LS^2's access model with a wider selection of permissions, and direct support for access control lists or more sophisticated methods of enforcing access control.

Isolation. Third, we want to develop techniques for reasoning about isolated process execution, which is quite common in systems. For example, when a machine is reset, rebooting is a purely sequential process until the operating system enables time sharing. Currently, isolated process execution has to be modeled by explicitly giving exclusive-write locks on all cells to the executing process, so that other concurrent processes have no effect on it. Although technically sound, this is artificial, and we would like to add axioms and rules to directly reason in these cases. In a related area, we want to enrich the programming model with dynamic creation of processes (a "fork" primitive), which is not possible hitherto.

Composition. Finally, in line with our focus on keeping reasoning as simple as possible, we are developing techniques to compose proofs modularly. We want to able to answer questions like the following: if system A provides property P_A and system B provides property P_B, then what meaningful property (if any) is obtained by allowing A and B to execute simultaneously? Suitable answers to this and similar questions would enable modular reasoning about system components, and reuse of existing proofs.

References

[1] Advanced Micro Devices. AMD64 virtualization: Secure virtual machine architecture reference manual. AMD Publication no. 33047 rev. 3.01, May 2005.

[2] S. Brookes. A semantics for concurrent separation logic. In *Proceedings of 15th CONCUR*. 2004.

[3] E. M. Chan, et al. BootJacker: Compromising computers using forced restarts. In *Proceedings of 15th ACM CCS*. 2008.

[4] A. Datta, et al. Protocol Composition Logic (PCL). *Electr. Notes Theor. Comput. Sci.*, 172:311–358, 2007.

[5] S. Garriss, et al. Towards trustworthy kiosk computing. In *Workshop on Mobile Computing Systems and Applications*. Feb. 2006.

[6] D. Harel, D. Kozen, J. Tiuryn. *Dynamic Logic*. Foundations of Computing. MIT Press, 2000.

[7] C. A. R. Hoare. An axiomatic basis for computer programming. *CACM*, 12(10):576–580, 1969. ISSN 0001-0782.

[8] Intel Corporation. Trusted eXecution Technology – preliminary architecture specification and enabling considerations. Document number 31516803, Nov. 2006.

[9] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3), May 1994.

[10] R. Milner, M. Tofte, R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990. ISBN 0-262-63132-6.

[11] J. Saltzer, M. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, Sept. 1975.

[12] TCG. PC client specific TPM interface specification (TIS). Version 1.2, Revision 1.00, Jul. 2005.

[13] Trusted Computing Group (TCG). `https://www.trustedcomputinggroup.org/`, 2009.