# A Logic of Secure Systems and its Application to Trusted Computing

Anupam Datta
*danupam@cmu.edu*

Jason Franklin
*jfrankli@cs.cmu.edu*

Deepak Garg
*dg@cs.cmu.edu*

Dilsun Kaynar
*dilsun@cs.cmu.edu*

## Abstract

*We present a logic for reasoning about properties of secure systems. The logic is built around a concurrent programming language with constructs for modeling machines with shared memory, a simple form of access control on memory, machine resets, cryptographic operations, network communication, and dynamically loading and executing unknown (and potentially untrusted) code. The adversary's capabilities are constrained by the system interface as defined in the programming model (leading to the name* CSI-ADVERSARY*). We develop a sound proof system for reasoning about programs without explicitly reasoning about adversary actions. We use the logic to characterize trusted computing primitives and prove code integrity and execution integrity properties of two remote attestation protocols. The proofs make precise assumptions needed for the security of these protocols and reveal an insecure interaction between the two protocols.*

## 1. Introduction

Contemporary secure systems are complex and designed to provide subtle security properties in the face of attack. Examples of such systems include hypervisors, virtual machine monitors, security kernels, operating systems, web browsers, and secure co-processor-based systems such as those utilizing the Trusted Computing Group's Trusted Platform Module (TPM) [1]. In this paper we initiate a program to formally model abstractions of such systems and specify and analyze their security properties in the presence of a general class of adversaries. Specifically, we introduce the Logic of Secure Systems ($LS^2$) and use it to carry out a detailed analysis of Trusted Computing systems. The logic is built around a programming language for modeling systems and is inspired by a logic for network protocol analysis, Protocol Composition Logic (PCL) [2]–[5].

**Programming Model.** 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. Following PCL, the language includes process calculi and functional constructs for modeling cryptographic operations, straightline code, and network communication. We introduce 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 [6]. We model memory protection, a fundamental building block for secure systems [7], by allowing programs to acquire exclusive-write locks on memory locations. The treatment of dynamically loading and executing unknown code is novel to this work.

While these constructs are the common denominator for many secure systems, including the trusted computing systems examined in this paper, they are by no means sufficient to model all systems of interest. The language, however, is *extensible* in a modular fashion, as we illustrate by extending the core language (presented in Section 2) with a trusted computing subsystem (in Section 3). 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. Platform Configuration Registers (PCR) in the TPM are an example since they can be modeled as a special form of memory that may be accessed via read, reset, and a new extend operation. Some extensions can have a more global effect on the language semantics. For instance, adding the reset operation to the language affects both how state of local memory and TPM PCRs may be updated.

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 locations, but can only update PCRs 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 messages on the network, inject messages that it can create, read and write memory locations 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 to compromise systems.

**Logic.** Security properties of programs are expressed in $LS^2$ using modal formulas of the form $[P]_I^{t_b,t_e}\ A$, which means that formula $A$ holds whenever thread $I$ executes exactly the program $P$ in the time interval $(t_b, t_e]$, irrespective of the actions executed concurrently by other threads including the adversary. The thread $I$ identifies the principal executing the program, the machine on which the program is being executed, and includes a unique identifier. The formula $A$ expresses security properties, such as confidentiality, integrity, authentication, as well as code and execution integrity. The logic includes predicates that reflect the programming language constructs for shared memory, memory protection, machine resets and a form of unconditional jump to model branching to dynamically loaded code.

Security properties are established using a proof system for $LS^2$. A central design goal that $LS^2$ achieves (following PCL) is that *the proof system does not mention adversary actions*. Instead, the semantics and soundness of the proof system guarantee that if $[P]_I^{t_b,t_e} A$ is provable, then $A$ holds in all traces in which $I$ completes execution of program $P$, including those that contain adversarial threads. This implicit treatment of adversaries simplifies proofs significantly. Designing a sound proof system that supports this local style of reasoning, in spite of the global nature of shared memory changes and execution of dynamically loaded unknown code, turned out to be a significant technical challenge.

We formalize local reasoning principles about shared memory with axioms that reason about invariance of values in memory based on local actions of threads that hold locks (see Section 2). This approach is technically similar to concurrent separation logic, whose regions resemble $LS^2$'s locks [8], but distinct from formal systems which support global reasoning about concurrent shared memory programs [9]. Our initial idea to reason about execution of dynamically loaded code was to treat the code being branched to as a continuation of the code calling it. However, this approach does not work for the case where the code being branched to is either read from memory or received over the network, because nothing can be determined about the called code by looking at the caller's program. As a result, traditional methods for proving program invariants such as those based on Hoare logic and its extensions [10]–[12] do not apply to this setting. Yet this is exactly what we needed to reason in the face of adversaries who can modify or inject code into the system. Our final technical approach for reasoning about execution of dynamically loaded code is based on a program invariance rule, which we elaborate on in Section 2 and illustrate in Section 4.1.

**Trusted Computing.** We model and analyze two trusted computing protocols that rely on TPMs to provide integrity properties: load-time attestation using a Static Root of Trust for Measurement (SRTM) [13] and late-launch-based attestation using a Dynamic Root of Trust for Measurement (DRTM) [14]–[16]. In doing so, we make the following contributions. First, we formalize, using axioms, the behavior of core trusted computing primitives including the TCG's widely-deployed secure co-processor, the Trusted Platform Module (TPM), as well as recently introduced hardware to support the *late launch* of a security kernel in a protected execution environment. Hardware implementations of late launch are publicly available in both AMD's Secure Virtual Machine Architecture (SVM) [15] and Intel's Trusted eXecution Technology (TXT) [16]. These axioms provide a succinct specification of the primitives, which serve as building blocks in the proofs of the protocols (see Section 3).

Second, we formally define and prove code integrity and execution integrity properties of the attestation protocols (Section 4; Theorems 2–4). To the best of our knowledge, these are the first logical security proofs of these protocols.

Finally, the formal proofs yield insights about the security of these protocols. The invariants used in the proofs make precise the properties that the Trusted Computing Base (TCB) must satisfy. In Section 4, we describe these invariants and manually check that an invariant holds on a security kernel implementation used in an attestation protocol. We demonstrate that newly introduced hardware support for late launch actually adversely affects the security of previous generation attestation protocols. We describe an attack that utilizes hardware support for late launch to exploit load-time attestation protocols that measure software starting at system boot. The attack enables an adversary to report false system integrity measurements that are not tied to the actual state of the platform. This attack could be used to exploit Digital Rights Management (DRM) protocols that rely on load-time attestation.

## 2. Logic of Secure Systems

We introduce the syntax of the Logic of Secure Systems ($LS^2$) in this section. The next section introduces features of $LS^2$ that are specific to trusted computing. Due to lack of space, we restrict technical descriptions to the extent necessary to explain the main concepts and application, and refer the reader to a technical report for details [17].

| | | | | |
|---|---|---|---|---|
| Expressions/Values | $e$ | ::= | $n$ | Number |
| | | $\mid$ | $\hat{X},\hat{Y}$ | Agent |
| | | $\mid$ | $K$ | Key |
| | | $\mid$ | $K^{-1}$ | Inverse of key $K$ |
| | | $\mid$ | $x$ | Variable |
| | | $\mid$ | $(e,e')$ | Pair |
| | | $\mid$ | $SIG_K\{\lvert e\rvert\}$ | Value $e$ signed by private key $K$ |
| | | $\mid$ | $ENC_K\{\lvert e\rvert\}$ | Value $e$ encrypted by public key $K$ |
| | | $\mid$ | $SYMENC_K\{\lvert e\rvert\}$ | Value $e$ encrypted by symmetric key $K$ |
| | | $\mid$ | $H(e)$ | Hash of $e$ |
| | | $\mid$ | $P$ | Program reified as data |
| Machine | $m$ | | | |
| Location | $l$ | ::= | $m.RAM.k \mid m.disk.k \mid m.pcr.k \mid m.dpcr.k$ | |
| Action | $a$ | ::= | `read` $l$ | Read location $l$ |
| | | $\mid$ | `write` $l,e$ | Write $e$ to location $l$ |
| | | $\mid$ | `extend` $l,e$ | Extend PCR $l$ with $e$ |
| | | $\mid$ | `lock` $l$ | Obtain write lock on location $l$ |
| | | $\mid$ | `unlock` $l$ | Release write lock on location $l$ |
| | | $\mid$ | `send` $e$ | Send $e$ as a message |
| | | $\mid$ | `receive` | Receive a message |
| | | $\mid$ | `sign` $e,K$ | Sign $e$ with private key $K$ |
| | | $\mid$ | `verify` $e,K$ | Check that $e = SIG_K\{\lvert e'\rvert\}$ |
| | | $\mid$ | `enc` $e,K$ | Encrypt $e$ with public key $K$ |
| | | $\mid$ | `dec` $e,K$ | Decrypt $e$ with private key $K$ |
| | | $\mid$ | `symenc` $e,K$ | Encrypt $e$ with symmetric key $K$ |
| | | $\mid$ | `symdec` $e,K$ | Decrypt $e$ with symmetric key $K$ |
| | | $\mid$ | `hash` $e$ | Hash the expression $e$ |
| | | $\mid$ | `eval` $f,e$ | Evaluate function $f$ with argument $e$ |
| | | $\mid$ | $\mathtt{proj}_1$ $e$ | Project the 1st component of a pair |
| | | $\mid$ | $\mathtt{proj}_2$ $e$ | Project the 2nd component of a pair |
| | | $\mid$ | `match` $e,e'$ | Check that $e = e'$ |
| | | $\mid$ | `new` | Generate a new nonce |
| Program | $P,Q$ | ::= | $\cdot \mid$ `jump` $e$ $\mid$ `latelaunch` $\mid x := a;P$ | |
| Thread id | $I,J$ | ::= | $\langle \hat{X},\eta,m\rangle$ | |
| Thread identifier | $\eta$ | | | |
| Thread | $T,S$ | ::= | $[P]_I$ | |
| Store | $\sigma$ | : | Locations → Expressions | |
| Lock map | $\iota$ | : | Locations → (Thread ids) $\cup$ $\{\_\}$ | |
| Configuration | $C$ | ::= | $\iota,\sigma,T_1 \mid \ldots \mid T_n$ | |

Figure 1. Syntax of the programming language

## 2.1. Programming Model

The programming language definition includes its syntax and operational semantics. The syntax is summarized in Figure 1. The current language includes process calculi and functional constructs for modeling cryptographic operations, straightline code, and network communication among concurrent processes, but does not have conditionals (if…then…else…), returning function calls or loops. Instead, it has a match construct that tests equality of expressions (`match` $e,e'$) and blocks if the test fails, as well as unconditional jumps to arbitrary code (`jump` $e$). These constructs are sufficient for applications we have considered so far. In future work, we plan to investigate the technical challenges associated with adding conditionals, returning function calls, and loops to the language. We describe below the core language constructs, the adversary model, and the form of the operational semantics. Examples of programs in the language can be found in Section 4.

**Data, agents, and keys.** Data is represented in the programming model symbolically as expressions $e$ (also called values). Expressions may be numbers $n$, identities of agents (principals) $\hat{X}$, keys $K$, variables $x$, pairs $(e,e')$, signatures using private keys $SIG_K\{|e|\}$ (denoting the signature on $e$ made using the key $K$), asymmetric key encryptions $ENC_K\{|e|\}$, symmetric key encryptions $SYMENC_K\{|e|\}$, hashes $H(e)$, or code reified as data $P$. All expressions are assumed to be simply typed (e.g. a pair can be distinguished from a number), but we elide the details of the types. Agents, denoted $\hat{X},\hat{Y}$, are users associated with a system on behalf of whom programs execute. Keys are denoted by the letter $K$. The inverse of key $K$ is denoted by $K^{-1}$. We assume that the expression $e$ may be recovered from the signature $SIG_K\{|e|\}$ if the verification key $K$ is known. We also assume that hashes are confidentiality preserving.

**Systems, programs, and actions.** A *secure system* is specified as a set of programs $P$ in the programming language. For example, a trusted computing attestation system will contain two programs, one to be executed by the untrusted platform and the other by the remote verifier. Each *program* consists of a number of actions $x := a$ that are executed in a straight line. The name $x$ binds the value returned by the action $a$, and is used to refer to the value in subsequent actions. Our model of straightline code execution is thus functional. This design choice simplifies reasoning significantly. For some actions such as sending a message, the value returned is meaningless. In such cases we assume that the value returned is the constant 0. A program ends with either an empty action $\cdot$, or one of the special actions `jump` $e$ or `latelaunch` . The expression `jump` $e$ is described below and `latelaunch` is covered in the next section. A single executing program is called a *thread* $[P]_I$ (threads are referred to with variables $T$, $S$). It contains a program $P$, and a descriptor $I$ for the thread that is a tuple $\langle \hat{X}, \eta, m \rangle$. $\hat{X}$ is the agent that owns the thread, $m$ is the machine on which the thread is hosted, and $\eta$ is a unique identifier (akin to a process id). The abstract runtime environment of the language is called a *configuration* $C$, written $\iota, \sigma, T_1 | \ldots | T_n$. It contains all executing threads ($T_1 | \ldots | T_n$), the state of memory on all machines (represented by the map $\sigma$), and the state of memory locks held by threads (represented by the map $\iota$).

**Cryptography and network primitives.** The programming language includes actions for standard operations like signing and signature verification, encryption and decryption (both symmetric and asymmetric), nonce generation, hashing, expression matching, projection from a pair, and evaluation of arbitrary side-effect free functions (`eval` $f, e$). Threads can communicate with each other 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 cryptography and network communication follows PCL. The language constructs we present next are new to this work.

**Machines and shared memory.** Threads can also share data through memory. The programming model contains machines $m$ explicitly. Each machine contains a number of memory locations $l$ that are shared by all threads running on the machine. Each location is classified as either RAM, persistent store (hard disk), or other special purpose location (such as Platform Configuration Registers that are described in the next section). The machine on which a location exists and the location's type are made explicit in the location's name. For instance, $m.RAM.k$ is the $k$th RAM location on machine $m$. The behavior of a location depends on its type. For example, RAM locations are set to a fixed value when a machine resets, whereas persistent locations are not affected by resets. Despite these differences, the prominent characteristics of all locations 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 location.

**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 location $l$ by executing the action `lock` $l$. Information on locks held by threads is included in a configuration as a map $\iota$ from locations to identities of threads that hold locks on them. The semantics of the programming language guarantee that while a lock is held by a thread, no other thread will be able to write the location. A thread may relinquish a lock it holds by executing the action `unlock` $l$. Locking in this manner may be used to enforce integrity of contents of memory. Similarly, one may add read locks that provide confidentiality of memory contents. Although technically straightforward, read locks are omitted from this paper since we are focusing on integrity properties.

**Machine resets.** The language allows a machine to be spontaneously reset. There is no specific action that causes a reset. Instead, there is a reduction in the operational semantics that may occur at any time to reset a machine. When this happens, all running threads on the machine are killed, all its RAM and PCR locations 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 [18]. In the context of trusted computing, e.g., the fact that a TPM's Platform Configuration Registers (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 [19].

**Untrusted code execution.** The last salient feature of our programming model is an action `jump` $e$ that dynamically branches to code represented by the expression $e$. The code $e$ is arbitrary; it may have been read from memory or disk, or even have been received over the network. As a result, it 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.

**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 she break cryptography.

**Operational semantics.** 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. A *trace* $C_0 \longrightarrow C_1 \ldots \longrightarrow C_n$ is a sequence of configurations, such that successive configurations in the sequence can be obtained by applying one reduction rule. A *timed trace* $C_0 \xrightarrow{t_1} C_1 \ldots \xrightarrow{t_n} C_n$ associates monotonically increasing time points $t_1, \ldots, t_n$ with reductions on a trace. These time points may be drawn from any totally ordered set, such as integers or real numbers.

## 2.2. Logic

The logic $LS^2$ is used to specify and reason about properties of secure systems.

**Syntax.** Figure 2 summarizes $LS^2$'s syntax, including predicates specific to trusted computing that we discuss in the next section. Predicates for representing network communication and cryptographic operations are taken from PCL. Other predicates that capture information about state, unconditional jumps, and resets are new to this work. A significant difference from PCL is that $LS^2$ incorporates time explicitly in formulas and semantics. All predicates and formulas are interpreted relative to not only a timed trace but also a point of time (modal formulas, described below, are an exception since they are interpreted relative to a timed trace only). In the proof system, time is used to track the relative order of actions on a trace and to specify program invariants.

*Action predicates* capture actions performed by threads. For instance, $\mathsf{Send}(I,e)$ holds on a trace at time $t$ if thread $I$ executes action `send` $e$ at time $t$ in the trace. $\mathsf{Write}(I,l,e)$ holds on a trace whenever thread $I$ executes `write` $l,e$. Similarly, we have predicates to capture cryptographic operations. *General predicates* capture other information, including information about the state of the environment. Particularly prominent are the two predicates $\mathsf{Mem}(l,e)$ which holds whenever location $l$ contains value $e$, and $\mathsf{Jump}(I,e)$ which holds whenever thread $I$ executes `jump` $e$. Access control on memory is reflected in the logic through three predicates: $\mathsf{Lock}(I,l)$, $\mathsf{Unlock}(I,l)$, and $\mathsf{IsLocked}(l,I)$. The first two of these capture actions: $\mathsf{Lock}(I,l)$ holds on a trace when a thread $I$ obtains an exclusive-write lock on location $l$, whereas $\mathsf{Unlock}(I,l)$ holds when thread $I$ releases the lock. The third predicate $\mathsf{IsLocked}(l,I)$ captures state: it holds whenever thread $I$ has an exclusive-write lock on location $l$. As an example, suppose that thread $I$ executes an action to obtain the lock on location $l$ at time $t$ and executes another action to release the lock at a later point $t'$. Then $\mathsf{Lock}(I,l)$ will hold exactly at time $t$, $\mathsf{Unlock}(I,l)$ will hold exactly at time $t'$, and $\mathsf{IsLocked}(l,I)$ will hold at all points of time between $t$ and $t'$. The predicate $\mathsf{Reset}(m,I)$ holds at time $t$ if machine $m$ is reset at time $t$, creating the new thread $I$ to boot it. We define the abbreviations $\mathsf{Reset}(m)$ and $\mathsf{Jump}(I)$ as $\exists I.\ \mathsf{Reset}(m,I)$ and $\exists e.\ \mathsf{Jump}(I,e)$ respectively. $\mathsf{Contains}(e,e')$ means that $e'$ is a sub-expression of $e$. The predicate $\mathsf{Honest}(\hat{X},\vec{P})$ is described in Section 3.1.

Predicates can be combined using the usual logical connectives: $\wedge$ (conjunction), $\vee$ (disjunction), $\supset$ (implication), and $\neg$ (negation) as well as first-order universal and existential quantifiers that may range over expressions, keys, principals, threads, locations, and time. There is a special formula, $A\ @\ t$, which captures time explicitly in the logic. $A\ @\ t$ means that formula $A$ holds at time $t$. We often write intervals in the usual mathematical sense; they may take the forms $(t_1,t_2)$, $[t_1,t_2]$, $(t_1,t_2]$, and $[t_1,t_2)$. For an interval $i$, we also define the formula $A$ on $i$ as $\forall t.\ ((t \in i) \supset A\ @\ t)$, where $t \in i$ is the obvious membership predicate. $A$ on $i$ means that $A$ holds at each point in the interval $i$. This treatment of time in the logic draws ideas from work on hybrid modal logic [20]–[22].

Security properties of programs are expressed in $LS^2$ using one of two forms of modal formulas. The principal of these, $[P]_I^{t_b,t_e}\ A$, means that formula $A$ holds whenever thread $I$ executes exactly the program $P$ sequentially in the semi-open interval $(t_b,t_e]$. $A$ may mention any variables occurring unbound in $P$. It usually expresses a safety property about the program $P$. For example, if $P$ is the client program of a key exchange protocol, $A$ may say that $P$ generated a key after $t_b$, sent it to a server, and received a confirmation that it was received. Examples of security properties for trusted computing systems can be found in Section 4.

**Proof System.** Security properties of a program are established using a proof system for $LS^2$. This proof system contains some basic rules for reasoning about modal formulas, and a number of axioms that capture intuitive properties of program behavior. Parts of the proof system, particularly the part dealing with cryptographic primitives were easily designed using existing ideas from PCL. As mentioned in the introduction, a central design goal that $LS^2$ achieves is that the proof system does not mention adversary actions. We elaborate below on the

| | | | |
|---|---|---|---|
| Action Predicates | $R$ | $::=$ | $\mathsf{Receive}(I,e) \mid \mathsf{Send}(I,e) \mid \mathsf{Sign}(I,e,K) \mid \mathsf{Verify}(I,e,K) \mid \mathsf{Encrypt}(I,e,K) \mid \mathsf{Decrypt}(I,e,K) \mid$ $\mathsf{SymEncrypt}(I,e,K) \mid \mathsf{SymDecrypt}(I,e,K) \mid \mathsf{Hash}(I,e) \mid \mathsf{Eval}(I,f,e,e') \mid \mathsf{Match}(I,e,e') \mid$ $\mathsf{New}(I,n) \mid \mathsf{Write}(I,l,e) \mid \mathsf{Read}(I,l,e) \mid \mathsf{Lock}(I,l) \mid \mathsf{Unlock}(I,l) \mid \mathsf{Extend}(I,l,e)$ |
| General Predicates | $M$ | $::=$ | $\mathsf{Mem}(l,e) \mid \mathsf{IsLocked}(l,I) \mid \mathsf{Reset}(m,I) \mid \mathsf{Jump}(I,e) \mid \mathsf{LateLaunch}(m,I) \mid \mathsf{Contains}(e,e') \mid$ $e = e' \mid t \geq t' \mid \mathsf{Honest}(\hat{X},\vec{P})$ |
| Formulas | $A,B$ | $::=$ | $R \mid M \mid \top \mid \bot \mid A \wedge B \mid A \vee B \mid A \supset B \mid \neg A \mid \forall x.A \mid \exists x.A \mid A \mathbin{@} t$ |
| Modal Formulas | $J$ | $::=$ | $[P]_I^{t_b,t_e}\ A \quad \mid \quad [a]_{I,x}^{t_b,t_e}\ A$ |

Figure 2. Syntax of $LS^2$

technical approach for designing a sound proof system that supports this local style of reasoning in spite of the global nature of shared memory changes and execution of dynamically loaded code.

We reason about memory locally using axioms that establish invariance of values in memory, using information about locks and actions of threads that hold the locks. These axioms are modular (there is one set of axioms for each type of memory) and extensible (more axioms can be added for new types of memory, as we do for Platform Configuration Registers in Section 3). As examples, the following two axioms are invariance rules for locations of RAM and disk respectively. The first axiom says that if location $m.RAM.k$ (denoting a location with address $k$ in the RAM of machine $m$) contains value $e$ at time $t_b$, during the interval $(t_b,t_e)$ thread $I$ has a lock on this location, thread $I$ does not write to the location, and machine $m$ is not reset during the interval, then $m.RAM.k$ must contain the value $e$ throughout the interval $(t_b,t_e)$. The second axiom is similar, but it applies to locations on disk. In this case, the precondition that machine $m$ not be reset is unnecessary because contents of the disk do not change due to a reset.

$$
\begin{array}{ll}
\text{(MemIR)} & \vdash (\mathsf{Mem}(m.RAM.k,e) \mathbin{@} t_b) \\
& \quad \wedge (\mathsf{IsLocked}(m.RAM.k,I) \text{ on } (t_b,t_e)) \\
& \quad \wedge (\forall e'.\ \neg\mathsf{Write}(I,m.RAM.k,e') \text{ on } (t_b,t_e)) \\
& \quad \wedge (\neg\mathsf{Reset}(m) \text{ on } (t_b,t_e)) \\
& \qquad \supset (\mathsf{Mem}(m.RAM.k,e) \text{ on } (t_b,t_e))
\end{array}
$$

$$
\begin{array}{ll}
\text{(MemID)} & \vdash (\mathsf{Mem}(m.disk.k,e) \mathbin{@} t_b) \\
& \quad \wedge (\mathsf{IsLocked}(m.disk.k,I) \text{ on } (t_b,t_e)) \\
& \quad \wedge (\forall e'.\ \neg\mathsf{Write}(I,m.disk.k,e') \text{ on } (t_b,t_e)) \\
& \qquad \supset (\mathsf{Mem}(m.disk.k,e) \text{ on } (t_b,t_e))
\end{array}
$$

For reasoning about execution of dynamically loaded code, we introduce the following rule that allows us to combine information about the invariants of a program $P$ with the knowledge that the program was branched to. We define a program invariant as a property that holds whenever any prefix of the sequence of actions of the program executes. The prefixes or initial sequences $IS(P)$ of a program $P$ are formally defined as follows: $IS(\cdot) = \{\cdot\}$, $IS(\texttt{jump } e) = \{\cdot, \texttt{jump } e\}$, $IS(\texttt{latelaunch }) = \{\cdot, \texttt{latelaunch }\}$, $IS(x := a;P) = \{\cdot\} \cup \{x := a;Q \mid Q \in IS(P)\}$.

$$
\frac{\begin{array}{c}\text{For every } Q \text{ in } IS(P): \quad \vdash [Q]_I^{t_b,t_e}\ A(t_b,t_e) \\ (t_b,t_e \text{ fresh constants})\end{array}}{\vdash \mathsf{Jump}(I,P) \mathbin{@} t \supset \forall t'.\ (t' > t) \supset A(t,t')}\,\text{Jump}
$$

In its premise the rule requires that for every initial sequence $Q$ of $P$, there be a proof, generic in the constants $t_b$ and $t_e$, that establishes $A(t_b,t_e)$ given that $Q$ executes in thread $I$ during the interval $(t_b,t_e]$. The conclusion says that if thread $I$ branches to program $P$ at time $t$ (assumption $\mathsf{Jump}(I,P) \mathbin{@} t$), then for any time $t' > t$, $A(t,t')$ must hold. Informally, we may explain the soundness of this rule as follows. If thread $I$ branches to code $P$ at time $t$, then for any $t' > t$, the thread $I$ must execute some prefix of $P$ in the interval $(t,t']$. Instantiating the premise with this prefix $Q$, and $t,t'$ for $t_b,t_e$, we get exactly the desired property $A(t,t')$.

The above rule is central among $LS^2$'s principles for reasoning about dynamically loaded code, which we believe to be novel. Both a discussion of the novelty and an example of the reasoning principles are postponed to Section 4.1. Whereas their application to reasoning about dynamically loaded code is new, invariants over initial segments of code are not a contribution of this work. PCL uses invariants similar to ours to reason about principals who are executing known pieces of code. $LS^2$ also uses invariants for many other purposes besides reasoning about jumps, including reasoning about resets. The latter is simpler than reasoning about jumps, because we assume that when a machine is reset, a *fixed* program is started to reboot the machine. The code marked $SRTM(m)$ in Figure 3 is one example of the form this program may have.

**Semantics and Soundness.** Formulas of $LS^2$ are interpreted over timed traces obtained from execution of a program in the programming language. The proof system of $LS^2$ is formally connected to the programming language semantics through a program independent *soundness theorem* which guarantees that any property established in the proof system actually holds over all traces obtainable from the program and any number of adversarial threads. Let $\Gamma$ denote a set of formulas, and $\varphi$ denote a formula or a modal formula. Further, let $\Gamma \vdash \varphi$ denote provability in $LS^2$'s proof system, and $\Gamma \models \varphi$ denote semantic entailment. Our main technical result for $LS^2$ is the following soundness theorem.

**Theorem 1** (Soundness)**.** *If* $\Gamma \vdash \varphi$ *then* $\Gamma \models \varphi$.

The proof of this theorem, as well as those of all later theorems, can be found in the full version of this paper [17].

## 3. Modeling Trusted Computing Primitives

This section describes extensions to $LS^2$ to model and reason about hardware primitives used with protocols specified by the Trusted Computing Group (TCG). These hardware primitives include the TCG's Trusted Platform Module (TPM) and static Platform Configuration Registers (PCRs), as well as the more recent hardware support for late launch and dynamic PCRs as implemented by AMD's Secure Virtual Machine (SVM) extensions [23] and Intel's Trusted eXecution Technology (TXT) [16]. We describe below the hardware primitives and their formalization in $LS^2$ at a high level. In subsequent sections, we use our formalizations to prove security properties of trusted computing protocols.

### 3.1. Trusted Platform Module

The Trusted Platform Module (TPM) is a secure coprocessor that performs cryptographic operations such as encryption, decryption, and creation and verification of digital signatures. Each TPM includes a unique embedded private key (called the Attestation Identity Key or AIK). The public key corresponding to each AIK is published in a manufacturer-signed certificate. The private component of the AIK is assumed to be protected from compromise by malicious software. As a result, signatures produced by a TPM are guaranteed to be authentic, and unique to the platform on which the TPM resides.

We model relevant aspects of the TPM in $LS^2$ as follows. The private attestation identity key of the TPM on machine $m$ is modeled as a value in $LS^2$, denoted $AIK^{-1}(m)$. Its corresponding public key is denoted $AIK(m)$. The TPM itself is represented as a principal, denoted $\hat{AIK}(m)$. Of the many programs hardcoded into the TPM, only two are relevant for our purposes. These are idealized by the $LS^2$ programs marked $TPM_{SRTM}(m)$ and $TPM_{DRTM}(m)$ in Figures 3 and 4 respectively, and are explained in the next section. Both the fact that the TPM executes only one of these programs, and the fact that the TPM's private key cannot be leaked are modeled in $LS^2$ by a single predicate:

$$\mathsf{Honest}(\hat{AIK}(m), \{TPM_{SRTM}(m), TPM_{DRTM}(m)\})$$

This predicate entails (through the rules and axioms of the proof system) that any signature created by the key $AIK^{-1}(m)$ could only have been created in the TPM on machine $m$. It can also be used to prove invariants about threads which are known to execute on the TPM, using a rule similar to (Jump) that was described in Section 2.2.[1] We emphasize that the predicate mentioned above is not an axiom in $LS^2$, since its soundness cannot be established directly. Instead, we always assume it explicitly when we reason about the TPM.

**Static PCRs.** Static Platform Configuration Registers (PCRs) are protected registers contained in every TPM. From our perspective, the relevant property of PCRs is that their contents can only be modified in two ways: (a) by resetting the machine on which the TPM resides; this sets all the static PCRs to a special value that we denote symbolically using the name *sinit* (*sinit* is zero on most platforms), and (b) through a special TPM interface `extend` , which takes two arguments: a PCR to modify, and a value $v$ that is appended to the PCR. Since each PCR is of a fixed length but may be asked to store arbitrarily many values, `extend` replaces the current value of the PCR with a hash of the concatenation of its current value and a hash of $v$. In pseudocode, the effect of extending PCR $p$ with value $v$ may be described as the assignment $p \leftarrow H(p \mid\mid H(v))$, where $\mid\mid$ denotes concatenation and $H$ denotes a hash function. More generally, if the values extended into a PCR after a reset are $v_1, \ldots, v_n$ in sequence, its contents will be $H(\ldots(H(sinit||H(v_1))||H(v_2))\ldots||H(v_n))$. We use the notation $seq(sinit, v_1, \ldots, v_n)$ to denote this value. A common use for PCRs is to extend integrity measurements of program code into them during the boot process, then to have the TPM sign them with its AIK, and to submit this signed aggregate to a remote party as evidence that the values were generated in sequence on the machine.

We model PCRs as a special class of memory in $LS^2$. The $k$th static PCR on machine $m$ is denoted $m.pcr.k$. PCRs can be read using the usual read action in $LS^2$'s programming language, and they can be locked for access control, but the usual write action does not apply to them. Instead, the `extend` program is modeled as a primitive action in the programming language. It has exactly the effect described in the previous paragraph. Properties of PCRs are captured through axioms in $LS^2$. For example, the following axiom models the fact that *sinit* is written to every PCR when a machine is reset. In words, it states that if machine $m$ is reset at time $t$, then any PCR $k$ on $m$ contains value *sinit* at time $t$.

$$\text{(MemPR)} \quad \vdash (\mathsf{Reset}(m) \ @\ t) \supset (\mathsf{Mem}(m.pcr.k, sinit) \ @\ t)$$

Several other important properties of PCRs arise as a consequence of their restricted interface. First, if a PCR contains *sinit* at time $t$, then the machine $m$ on which it

[1] The predicate Honest is adapted from a predicate of the same name in PCL. PCL's predicate is slightly weaker since it lacks the second argument, but the reasoning principles associated with the two are similar.

resides must have been reset *most recently* at some time $t'$ since a reset is the only way to put *sinit* into a PCR. This is captured by the following axiom:

$$\begin{aligned}\text{(PCR2)}\quad \vdash\ & (\mathsf{Mem}(m.pcr.k, sinit) \mathbin{@} t)\\ & \supset (\exists t'.\ (t' \leq t) \wedge (\mathsf{Reset}(m) \mathbin{@} t')\\ & \quad \wedge (\neg \mathsf{Reset}(m) \text{ on } (t', t]))\end{aligned}$$

Second, if a PCR contains $seq(sinit, v_1, \ldots, v_n)$ at time $t$, it must also have contained $seq(sinit, v_1, \ldots, v_{n-1})$ at some prior time $t'$, without any reset in the interim. Thus the contents of a PCR are witness to every extension performed on it since its last reset. Formally, this property is captured in $LS^2$ by the following axiom:

$$\begin{aligned}\text{(PCR1)}\quad \vdash\ & (\mathsf{Mem}(m.pcr.k, seq(sinit, v_1, \ldots, v_n)) \mathbin{@} t)\\ & \supset (\exists t'.\ (t' < t)\\ & \wedge (\mathsf{Mem}(m.pcr.k, seq(sinit, v_1, \ldots, v_{n-1})) \mathbin{@} t')\\ & \wedge (\neg \mathsf{Reset}(m) \text{ on } (t', t])) \qquad (n \geq 1)\end{aligned}$$

In many cases of interest, we need to prove that the value in a PCR does not change over a period of time. To this end, we introduce an invariance axiom for PCRs, similar to axioms (MemIR) and (MemID) from Section 2.2. The modular design of the logic eases the introduction of this axiom.

$$\begin{aligned}\text{(MemIP)}\quad \vdash\ & (\mathsf{Mem}(m.pcr.k, e) \mathbin{@} t_b)\\ & \wedge (\mathsf{IsLocked}(m.pcr.k, I) \text{ on } (t_b, t_e))\\ & \wedge (\forall e'.\ \neg \mathsf{Extend}(I, m.pcr.k, e') \text{ on } (t_b, t_e))\\ & \wedge (\neg \mathsf{Reset}(m) \text{ on } (t_b, t_e))\\ & \quad \supset (\mathsf{Mem}(m.pcr.k, e) \text{ on } (t_b, t_e))\end{aligned}$$

### 3.2. Late Launch and Dynamic PCRs

Another hardware feature available in trusted computing platforms is late launch. Late launch provides the ability to measure and invoke a program, typically a security kernel or Virtual Machine Monitor (VMM), in a protected environment. Upon receiving a late launch instruction (SKINIT on the AMD SVM and SENTER on the Intel TXT), the processor switches from the currently executing operating system to a Dynamic Root of Trust for Measurement (DRTM) from which it is possible to later resume the suspended operating system. The program to be executed in a late launch session is specified by providing the physical address of the Secure Loader Block (SLB). When a late launch is performed, interrupts are disabled, direct memory access (DMA) is disabled to all physical memory pages containing the SLB and debugging access is disabled. The processor then jumps to the code in the SLB. This code may load other code. In addition to providing a protected environment, a special set of PCRs called *dynamic* PCRs are reset with a special value that we call *dinit* symbolically and the code in the SLB is hashed and extended into the dynamic PCR 17 (*dinit* is distinct from *sinit*). The dynamic PCRs can then be extended with other values, and the contents of the PCRs, signed by the TPM's key AIK, can be submitted as evidence that a late launch was performed.

We formally model late launch by adding a new action `latelaunch` to $LS^2$'s programming language. This action can be executed by any thread. The operational semantics of the language are extended to ensure that whenever `latelaunch` executes a new thread $I$ is created with a special program $LL(m)$, which extends the SLB into a dynamic PCR and branches to it. This program is shown in Figure 4. Protection of $I$ is modeled using locks – when started, $I$ is given locks to all dynamic PCRs on the machine $m$ it uses. $I$ may subsequently acquire more locks to protect itself. In the logic, the implicit locking of dynamic PCRs is captured by the following axiom, which means that if some thread executes `latelaunch` on machine $m$ at time $t$, creating the thread $I$, then $I$ has a lock on any dynamic PCR on $m$ at time $t$. $m.dpcr.k$ denotes the $k$th dynamic PCR on machine $m$.

$$\begin{aligned}\text{(LockLL)}\quad \vdash\ & (\mathsf{LateLaunch}(m, I) \mathbin{@} t)\\ & \supset (\mathsf{IsLocked}(m.dpcr.k, I) \mathbin{@} t)\end{aligned}$$

Dynamic PCRs have properties very similar to static PCRs. For example, the following axiom, similar to (MemPR) described above, means that *dinit* is written to every dynamic PCR when a late launch happens.

$$\begin{aligned}\text{(MemLL)}\quad \vdash\ & (\mathsf{LateLaunch}(m, I) \mathbin{@} t)\\ & \supset (\mathsf{Mem}(m.dpcr.k, dinit) \mathbin{@} t)\end{aligned}$$

Axioms corresponding to (PCR1) and (PCR2) are also sound for dynamic PCRs. The difference is that Reset and *sinit* must be replaced by LateLaunch and *dinit* respectively. The following axiom is used to prove invariance properties of dynamic PCRs.

$$\begin{aligned}\text{(MemIdP)}\quad \vdash\ & (\mathsf{Mem}(m.dpcr.k, e) \mathbin{@} t_b)\\ & \wedge (\mathsf{IsLocked}(m.dpcr.k, I) \text{ on } (t_b, t_e))\\ & \wedge (\forall e'.\ \neg \mathsf{Extend}(I, m.dpcr.k, e') \text{ on } (t_b, t_e))\\ & \wedge (\neg \mathsf{Reset}(m) \text{ on } (t_b, t_e))\\ & \wedge (\neg \exists I.\ \mathsf{LateLaunch}(m, I) \text{ on } (t_b, t_e))\\ & \quad \supset (\mathsf{Mem}(m.dpcr.k, e) \text{ on } (t_b, t_e))\end{aligned}$$

## 4. Trusted Computing Protocols

We analyze two trusted computing protocols that rely on TPMs to provide integrity properties: load-time attestation using an SRTM and late-launch-based attestation using a DRTM. In an attestation protocol, a platform utilizes a TPM to attest to platform state by performing two steps: integrity measurement and integrity reporting. Integrity measurement consists of collecting cryptographic hashes of local software events such as program loading. Integrity reporting consists of transmitting collected measurements in a signed aggregate to an external verifier. The external verifier may then use the measurements to make trust

```
$SRTM(m)$        $\equiv$  $b =$ read $m.bl\_loc$;
                           extend $m.pcr.s, b$;
                           jump $b$

$BL(m)$          $\equiv$  $o =$ read $m.os\_loc$;
                           extend $m.pcr.s, o$;
                           jump $o$

$OS(m)$          $\equiv$  $a =$ read $m.app\_loc$;
                           extend $m.pcr.s, a$;
                           jump $a$

$APP(m)$         $\equiv$  $\ldots$

$TPM_{SRTM}(m)$  $\equiv$  $w =$ read $m.pcr.s$;
                           $r =$ sign $(PCR(s), w), AIK^{-1}(m)$;
                           send $r$

$Verifier(m)$    $\equiv$  $sig =$ receive ;
                           $v =$ verify $sig, AIK(m)$;
                           match $v, (PCR(s),$
                                  $seq(sinit, BL(m), OS(m), APP(m)))$
```

Figure 3. Security Skeleton for SRTM Attestation Protocol

decisions. We first analyze an attestation protocol using a Static Root of Trust for Measurement (SRTM), then we consider an attestation protocol utilizing hardware support for late launch and a Dynamic Root of Trust for Measurement (DRTM). We simplify both protocols by assuming the AIK has been certified as authentic by a manufacturer certificate and by verifying a fixed sequence of system integrity measurements.

## 4.1. Attestation Using a Static Root of Trust

We start by performing an analysis of a load-time attestation protocol using an SRTM. The security skeleton of the protocol is specified in Figure 3. A security skeleton retains only relevant actions, in this case, actions performing integrity measurement and reporting. The SRTM protocol is composed of code that performs measurement followed by code that performs integrity reporting. We analyze the components separately.

**4.1.1. Integrity Measurement.** In the SRTM protocol, integrity measurement starts after a machine reset. The programs marked $SRTM(m)$, $BL(m)$, and $OS(m)$ in Figure 3 represent those portions of the SRTM, boot loader, and operating system that participate in the measurement process. The $SRTM(m)$ program is always the first program invoked when a machine reboots. It first reads the boot loader's code $b$ from the fixed disk address $m.bl\_loc$, then measures the code by extending it into a static PCR $m.pcr.s$ (which in this case stores all measurements), and then branches to the the boot loader by executing the instruction `jump` $b$. The boot loader ($BL(m)$) in turn reads the operating system's code $o$ from a fixed location $m.os\_loc$, extends it into PCR $m.pcr.s$, and branches to it. The operating system ($OS(m)$) performs similar actions with the application's code $a$. The application ($APP(m)$) may perform any actions. In practice, the sequence of measurement and loading may continue beyond the first application but we have chosen to terminate it here because extending the chain further does not lead to any new insights about the security of the system.

**Security Property.** We summarize the integrity measurement security property as follows: if $m.pcr.s$ is protected while a machine boots, and the contents of $m.pcr.s$ are $seq(sinit, BL(m), OS(m), APP(m))$, then the initial software loaded on machine $m$ since its last reboot was $BL(m)$ followed by $OS(m)$. We now state this property formally. We define the formulas $\mathsf{ProtectedSRTM}(m)$ and $\mathsf{MeasuredBoot}_{\mathsf{SRTM}}(m,t)$ as follows.

$$\begin{aligned}
&\mathsf{ProtectedSRTM}(m) = \\
&\quad \forall t, I.\ (\mathsf{Reset}(m,I)\ @\ t) \supset (\mathsf{IsLocked}(m.pcr.s, I)\ @\ t)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{MeasuredBoot}_{\mathsf{SRTM}}(m,t) = \\
&\quad \exists t_T.\ \exists t_B.\ \exists t_O.\ \exists J.\ (t_T < t_B < t_O < t) \wedge \\
&\qquad (\mathsf{Reset}(m,J)\ @\ t_T) \wedge (\mathsf{Jump}(J, BL(m))\ @\ t_B) \wedge \\
&\qquad (\mathsf{Jump}(J, OS(m))\ @\ t_O) \wedge (\neg\mathsf{Reset}(m)\ \mathsf{on}\ (t_T, t]) \\
&\qquad (\neg\mathsf{Jump}(J)\ \mathsf{on}\ (t_T, t_B)) \wedge (\neg\mathsf{Jump}(J)\ \mathsf{on}\ (t_B, t_O))
\end{aligned}$$

$\mathsf{ProtectedSRTM}(m)$ means that any thread $I$ created to boot machine $m$ after a reset obtains an exclusive-write lock on $m.pcr.s$. $\mathsf{MeasuredBoot}_{\mathsf{SRTM}}(m,t)$ identifies software events on $m$ such as the boot loader and operating system being branched to before time $t$. It comprises four facts: (1) There exists a time $t_T$ before $t$ at which $m$ was reset, creating a thread $J$ to boot the machine ($\mathsf{Reset}(m,J)\ @\ t_T$), (2) This thread $J$ branched to the programs $BL(m)$ and $OS(m)$ at later time points $t_B$ and $t_O$ ($\mathsf{Jump}(J, BL(m))\ @\ t_B$ and $\mathsf{Jump}(J, OS(m))\ @\ t_O$), (3) $J$ did not make any other jumps in the interim ($\neg\mathsf{Jump}(J)$ on $(t_T, t_B)$) and ($\neg\mathsf{Jump}(J)$ on $(t_B, t_O)$), and (4) Machine $m$ was not reset between $t_T$ and $t$ ($\neg\mathsf{Reset}(m)$ on $(t_T, t]$). Equivalently, after its last reboot before time $t$, the first programs loaded on $m$ were $BL(m)$ and $OS(m)$. We believe this is a natural property to expect from a system integrity measurement protocol.

The following theorem formalizes our security property. It states that under the assumptions that $m.pcr.s$ is protected during booting, and that $m.pcr.s$ contains $seq(sinit, BL(m), OS(m), APP(m))$ at time $t$, it is guaranteed that the boot loader and operating system used to boot the machine are $BL(m)$ and $OS(m)$ respectively.

**Theorem 2** (Security of Integrity Measurement)**.** *The* *following is provable in $LS^2$:*

$$\begin{array}{l}\mathsf{ProtectedSRTM}(m) \vdash \\ \quad \mathsf{Mem}(m.pcr.s, seq(sinit, BL(m), OS(m), APP(m))) \text{ @ } t \\ \qquad \supset \mathsf{MeasuredBoot}_{\mathsf{SRTM}}(m,t)\end{array}$$

We refer the reader to the full version of this paper for a detailed proof of this theorem [17]. Major steps in the proofs are discussed below to illustrate novel reasoning principles in $LS^2$. All programs mentioned below refer to Figure 3.

(1) Using axioms (PCR1) and (PCR2) in succession on the antecedent $\mathsf{Mem}(m.pcr.s, seq(sinit, BL(m), OS(m), APP(m)))$ @ $t$, we show that all sub-sequences of $seq(sinit, BL(m), OS(m), APP(m))$ must have appeared in $m.pcr.s$ at times earlier than $t$, and that machine $m$ must have been reset at some time $t_T$, creating a thread $J$ to boot it. Formally, we obtain

$$\begin{array}{l}\exists t_T, t_1, t_2, t_3, J.\ (t_T \le t_1 < t_2 < t_3 < t) \\ \quad \wedge (\mathsf{Mem}(m.pcr.s, seq(sinit, BL(m), OS(m))) \text{ @ } t_3) \\ \quad \wedge (\mathsf{Mem}(m.pcr.s, seq(sinit, BL(m))) \text{ @ } t_2) \\ \quad \wedge (\mathsf{Mem}(m.pcr.s, sinit) \text{ @ } t_1) \\ \quad \wedge (\mathsf{Reset}(m,J) \text{ @ } t_T) \\ \quad \wedge (\neg \mathsf{Reset}(m) \text{ on } (t_T, t])\end{array}$$

(2) Since $m$ was reset creating thread $J$ (second from last conjunct above), it follows in our model that the thread $J$ above must have started with the program $SRTM(m)$. (We have omitted a description of the rules that force this to be the case.) Thus, we would like to proceed by proving an invariant of $SRTM(m)$. However, we can say *nothing* about the program $b$ loaded at the end of $SRTM(m)$. This is because $b$ is read from a memory location $m.bl\_loc$, which could potentially have been written by an adversarial thread earlier. Fortunately, the extension of $b$ into $m.pcr.s$ in the second line of $SRTM(m)$ lets us proceed. Precisely, this extension along with some basic properties of PCRs lets us prove the following property that is *parametric* (universally quantified) in the code $b$. $t_T$ and $J$ were obtained in property (1).

$$\begin{array}{l}\forall t', b, o. \\ (((\mathsf{Mem}(m.pcr.s, seq(sinit, b, o)) \text{ @ } t') \\ \wedge (t_T < t' \le t)) \\ \quad \supset \exists t_B.\ ((t_T < t_B < t') \wedge (\mathsf{Jump}(J, b) \text{ @ } t_B))) \\ \qquad \wedge (\mathsf{IsLocked}(m.pcr.s, J) \text{ @ } t_B))\end{array}$$

This property means that if at any time $t'$ between $t_T$ and $t$, $m.pcr.s$ contained $seq(sinit, b, o)$, then thread $J$ must have branched to $b$ at some time $t_B$ between $t_T$ and $t'$, and that $J$ must hold a lock on $m.pcr.s$ at $t_B$. Informally this holds because the action immediately following the extension in $SRTM(m)$ is `jump` $b$, so if there is a further extension with $o$, `jump` $b$ must have happened in the interim. The assumption $\mathsf{ProtectedSRTM(m)}$ is used to rule out the possibility that a thread other than $J$ extended $o$ into $m.pcr.s$ before `jump` $b$ happened, and to show that $J$ holds the lock on $m.pcr.s$ at $t_B$.

(3) We instantiate the property in (2), choosing $b = BL(m)$, $o = OS(m)$, and $t' = t_3$ ($t_3$ was obtained in (1)). Eliminating the antecedents of the implication using facts from (1), we obtain:

$$\begin{array}{l}\exists t_B.\ ((t_T < t_B < t_3) \wedge (\mathsf{Jump}(J, BL(m)) \text{ @ } t_B) \\ \qquad \wedge (\mathsf{IsLocked}(m.pcr.s, J) \text{ @ } t_B))\end{array}$$

(4) From (3) we know $\mathsf{Jump}(J, BL(m))$ @ $t_B$. Next we use the (Jump) rule from Section 2.2. In the premise we show that $\forall t_b, t_e.\ \forall Q \in IS(BL(m)).\ \vdash [Q]_J^{t_b,t_e}\ A(t_b, t_e)$ for a suitable invariant $A(t_b, t_e)$, whose details we omit here (see [17] for details). The main difficulty here is similar to that in (2): we do not know what $o$ in the program of $BL(m)$ may be. Again, the invariant we prove is parametric in $o$. Using the (Jump) rule, we obtain the following property.

$$\begin{array}{l}\forall t', o, a. \\ (((\mathsf{Mem}(m.pcr.s, seq(sinit, BL(m), o, a)) \text{ @ } t') \\ \wedge (t_B < t' \le t)) \\ \quad \supset \exists t_O.\ ((t_B < t_O < t) \wedge (\mathsf{Jump}(J, o) \text{ @ } t_O)))\end{array}$$

This property is very similar to that in (2), except that it follows from an invariant of $BL(m)$, not $SRTM(m)$. The fact $\mathsf{IsLocked}(m.pcr.s, J)$ @ $t_B$ from (3) is needed to rule out the possibility that a thread other than $J$ extended $a$ into $m.pcr.s$.

(5) We instantiate the property in (4), choosing $o = OS(m)$, $a = APP(m)$ and $t' = t$. Combining with facts from (1), we obtain:

$$\exists t_O.\ ((t_B < t_O < t) \wedge (\mathsf{Jump}(J, OS(m)) \text{ @ } t_O))$$

The facts $(\mathsf{Reset}(m,J) \text{ @ } t_T)$, $(\neg\mathsf{Reset}(m) \text{ on } (t_T, t])$, $(\mathsf{Jump}(J, BL(m)) \text{ @ } t_B)$, and $(\mathsf{Jump}(J, OS(m)) \text{ @ } t_O)$ in (1), (3), and (5) establish part of $\mathsf{MeasuredBoot}_{\mathsf{SRTM}}(m,t)$. The remaining part follows from a similar analysis with slightly stronger invariants in (2) and (4).

The hardest part in designing $LS^2$'s proof system was coming up with sound principles for reasoning about dynamically branching to unknown code that are illustrated above, and in particular, the (Jump) rule. Although the final design is simple to use, it was not obvious at first. We believe that this method for reasoning about branching to completely unknown code (like $b$ and $o$) is new to this work. Prior work on reasoning about dynamically loaded code, usually based on higher-order extensions of Hoare logic [24]–[28], assumes that at least the invariants of the code being branched to are known at the point of branch in the program. In our setting, this assumption is unrealistic because we allow executable code to either be obtained over the network or be read from memory, and hence, potentially, to come from an adversary.

**4.1.2. Insights From Analysis.** A number of insights follow from the analysis. These insights include highlighting an unexpected property, clarifying assumptions on the TCB, and identifying program invariants required for security.

**Property Excludes Last Jump.** A key insight from the analysis is that the integrity measurement protocol does not provide sufficient evidence to deduce that the last program in a chain of measurements is actually executed. For example, an adversary can reboot the platform after $OS(m)$ extends $APP(m)$, but before it is jumped to. Alternatively, a race may occur between two application-level processes whereby the OS extends the first into $m.pcr.s$ and then the other process reads the value in $m.pcr.s$ before the first process is branched to.

**TCB Assumptions.** The value of $m.pcr.s$ does not guarantee that the measured software was also executed unless it is also guaranteed that no other process had write access to $m.pcr.s$. If the latter assumption fails, an attack exists: a malicious process may extend a piece of code into $m.pcr.s$ without executing it. This assumption usually holds in practice because booting is generally single threaded, but may fail if for example a malicious thread executes on another processor core concurrently with the measurement thread. Formally, this shows up as the $\mathsf{ProtectedSRTM}(m)$ formula, which is a necessary assumption for the proof.

**Program Invariants.** To establish Theorem 2, we prove program invariants for the $SRTM(m)$ and $BL(m)$ programs. These invariants provide a specification of the properties that an SRTM and a boot loader program must satisfy to be secure in an integrity measurement protocol, i.e. the assumptions about the TCB. The $SRTM(m)$ invariant states that there exists a time point $t'$ and thread $J$ such that $J$ branches to the boot loader $b$, $J$ does not branch to any program at any time point before $t'$, $m.pcr.s$ contains the hashed value of the boot loader $b$, and $m.pcr.s$ is locked by $J$ at $t'$. The invariant of $BL(m)$ states that there exists a time point $t$ and thread $J$ such that $J$ branches to program code $o$ only after the entire program code $o$ has been measured into $m.pcr.s$. Kauer [29] performed a manual source code audit of a number of TPM-enabled boot loaders to check the informal security condition that "no code...is executed but not hashed." Our invariant on the boot loader $BL$ was developed independently during the course of proving the above theorem and is a formal specification of this condition. We envisage that these invariants can be used to derive properties to automatically check against implementations of TCB components, thereby providing greater assurance that the trusted components are trustworthy.

**4.1.3. Integrity Reporting.** After the integrity measurement protocol loads the PCRs with measurements, the measurements can be used by the TPM to attest to the identify of the software loaded on the local platform. This protocol, called integrity reporting, involves two participants. One of the participants is the remote party itself, called the verifier. Its code is marked $Verifier(m)$ in Figure 3. The other participant in the protocol is the TPM of machine $m$ in the role of $TPM_{SRTM}(m)$. This code is also shown in the same figure.

The integrity reporting protocol contains two steps. In the first, the TPM on machine $m$ reads the contents $w$ of $m.pcr.s$, signs them and an identifier (denoted $PCR(s)$) that uniquely identifies $m.pcr.s$ with its embedded private key $AIK^{-1}(m)$ and sends the signed aggregate to the remote verifier. In the second step, the remote verifier verifies this signature with the known public key $AIK(m)$, and checks that the contents of the signature match the pair of $PCR(s)$ and $seq(sinit, BL(m), OS(m), APP(m))$.

**Security Properties.** The security properties of integrity reporting are formalized by the following two $LS^2$ formulas, which we call $J_1$ and $J_2$ respectively.

$$[Verifier(m)]_V^{t_b,t_e}\ \exists t.\ (t < t_e) \wedge \\ (\mathsf{Mem}(m.pcr.s, seq(sinit, BL(m), OS(m), APP(m)))\ @\ t)$$

$$[Verifier(m)]_V^{t_b,t_e}\ \exists t.\ (t < t_e) \wedge \mathsf{MeasuredBoot}_{\mathsf{SRTM}}(m,t)$$

The first property ($J_1$) states that if the code $Verifier(m)$ is executed successfully between the time points $t_b$ and $t_e$, then there must be a time $t$ before $t_e$ at which $m.pcr.s$ contained $seq(sinit, BL(m), OS(m), APP(m))$. The second property ($J_2$) means that a remote verifier can identify the boot loader and operating system that were loaded on $m$ at some time prior to $t_e$.

To prove these properties, we require two new assumptions, which we combine in the set $\Gamma_{SRTM}$ below. The first of these assumptions states that the remote verifier is distinct from the TPM. This assumption is needed to distinguish protocol participants, and is true in practice. The second assumption is the honesty assumption for the TPM from Section 3.1 that guarantees that the TPM's signature cannot be forged, and that the TPM always executes only specified programs.

$$\begin{aligned}\Gamma_{SRTM} \ &= \\ &\{\hat{V} \neq AI\hat{K}(m), \\ &\quad \mathsf{Honest}(A\hat{I}K(m), \{TPM_{SRTM}(m), TPM_{DRTM}(m)\})\}\end{aligned}$$

**Theorem 3** (Security of Integrity Reporting)**.** *The following are provable in $LS^2$'s proof system:*

(1) $\Gamma_{SRTM} \vdash J_1$

(2) $\Gamma_{SRTM}, \mathsf{ProtectedSRTM}(m) \vdash J_2$

The proof of (1) critically relies on the assumption $\mathsf{Honest}(A\hat{I}K(m), \{TPM_{SRTM}(m), TPM_{DRTM}(m)\})$ to establish both that the TPM on $m$ actually produced a signature in the past, and that the value signed by the TPM was actually read from *m.pcr.s*. The latter follows from knowledge of the programs that the TPM may be executing. (2) follows from (1) and Theorem 2.

**4.1.4. Insights From Analysis.** The security analysis lead to a number of insights including highlighting weaknesses in the property provided by the protocol and identifying program invariants required for security.

**Staleness of Measurements.** A key insight from the analysis is that after executing the integrity reporting protocol, the verifier has no knowledge of how recent the time of measurement $t$ is in comparison to $t_e$, the time the verifier's execution finished. This staleness of measurements is inherent in the protocol: it is possible to reboot the machine with a different boot sequence after sending the signature to the remote verifier, as is known from prior work [19]. Formally, one can only prove that *m.pcr.s* contained the reported measurements at time $t$, but not after.

**Program Invariants.** In the process of proving the above theorem, we prove a program invariant for the roles of the TPM (i.e., $TPM_{SRTM}(m)$ and $TPM_{DRTM}(m)$). This invariant provides a specification of the properties that a TPM's signing role must satisfy. In particular, the invariant requires that if the TPM returns a value then the value is a signature over the value stored in *m.pcr.s* and that the TPM does not write to any memory locations. The latter constraint is necessary to prevent previously measured code from being modified after being measured.

## 4.2. Attestation Using a Dynamic Root of Trust

We perform an analysis of DRTM attestation using our model of hardware support for late launch. We jointly analyze the protocol code that performs integrity measurement and reporting.

**4.2.1. DRTM Protocol.** We describe the security skeleton of the DRTM attestation protocol in Figure 4. The DRTM protocol is a four agent protocol. The processes are: (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 reads the binary of the program $P(m)$ from the secure loader block (SLB), and measures then branches to $P(m)$, (3) $P(m)$ that measures the nonce, evaluates the function $f$ on input 0 (the function $f$ and its input may be changed depending on application), and extends a distinguished string $EOL$ into *m.dpcr.k* to signify the end of the late launch session. (4) $TPM_{DRTM}(m)$, 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 measurements, verifies the signature, and checks that the measurements match the sequence $(dinit, P(m), n, EOL)$.

**Security Property.** We summarize the DRTM 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 $TPM_{DRTM}(m)$ and $TPM_{SRTM}(m)$, then the remote verifier is guaranteed that $J$ performed a single late launch on machine $m$ at some time $t_L$, $J$ branched to $P(m)$ 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 some 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 $J_{DRTM}$ below.

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

In order to prove the property, we have to make the following assumptions.

$$
\begin{array}{l}
\Gamma_{DRTM} \;=\; \\
\quad \{\hat{V} \neq AI\hat{K}(m), \\
\qquad \mathsf{Honest}(A\hat{I}K(m), \{TPM_{SRTM}(m), TPM_{DRTM}(m)\})\}
\end{array}
$$

We also made the same assumptions in the SRTM protocol ($\Gamma_{SRTM} = \Gamma_{DRTM}$). We prove the following theorem:

**Theorem 4** (Security of DRTM)**.** *The following is provable in* $LS^2$*:* $\Gamma_{DRTM} \vdash J_{DRTM}$

As in the SRTM protocol, the security of the DRTM protocol relies on PCRs being append-only and write-protected in memory. In addition, the DRTM protocol relies on (1) write locks on all dynamic PCRs that are provided by the late launch and (2) a dynamic reset of *m.dpcr.k* , to reset the values in the dynamic PCRs to *dinit* and signal that $P(m)$ was executed with the protections provided by late launch.

```
$OS(m)$            $\equiv$  $n' =$ receive ;
                             write $m.nonce, n'$;
                             late_launch

$LL(m)$            $\equiv$  $P =$ read $m.SLB$;
                             extend $m.dpcr.k, P$;
                             jump $P$

$P(m)$             $\equiv$  $n'' =$ read $m.nonce$;
                             extend $m.dpcr.k, n''$;
                             eval $f, 0$;
                             extend $m.dpcr.k, EOL$

$TPM_{DRTM}(m)$    $\equiv$  $w =$ read $m.dpcr.k$;
                             $r =$ sign $(dPCR(k), w), AIK^{-1}(m)$;
                             send $r$

$Verifier(m)$      $\equiv$  $n =$ new ;
                             send $n$;
                             $sig =$ receive ;
                             $v =$ verify $sig, AIK(m)$;
                             match $v, (dPCR(k),$
                                   $seq(dinit, P(m), n, EOL))$
```

Figure 4. Security Skeleton for DRTM Attestation Protocol

**4.2.2. Insights From Analysis.** The security analysis lead to a number of insights including revealing an insecure protocol interaction between the DRTM and SRTM attestation protocols, highlighting differences with the SRTM protocol, and identifying program invariants required for DRTM security that we subsequently used to manually audit a security kernel implementation.

**Insecure Protocol Interaction.** In extending $LS^2$ to model DRTM, we discovered that adding late launch required us to weaken some axioms related to reasoning about invariance of values in memory in order to retain soundness in the proof system. With these weaker axioms, we were unable to prove the safety property of the SRTM protocol. Soon after, we realized that SRTM's safety property can actually be violated using `latelaunch` . Specifically, during the execution of the SRTM protocol, a late launch instruction may be issued by another thread before $OS(m)$ has been extended into $m.pcr.s$. The invoked program may then extend the code of the programs $OS(m)$ and $APP(m)$ into $m.pcr.s$ without executing them, and send signed measurements to the remote verifier. Since the contents of $m.pcr.s$ would be the sequence $seq(sinit, BL(m), OS(m), APP(m))$, the remote verifier would believe incorrectly that $OS(m)$ was executed and the SRTM protocol would fail to provide its expected integrity property. This vulnerability can be countered if the program loaded in a DRTM session were unable to change the contents of $m.pcr.s$ if SRTM were executing in parallel. In the final design of our formal model, we force this to be the case by letting the thread booting a machine to retain an exclusive-write lock on $m.pcr.s$ even in the face of a concurrent late launch, thus allowing a proof of correctness of SRTM.

Late launch also opens the possibility of a code modification attack on SRTM. Specifically, after the code of a program such as $BL(m)$ or $OS(m)$ has been extended into $m.pcr.s$ in SRTM, a concurrent thread may invoke a DRTM session and change the code in memory before it is executed. Any subsequent attestation of integrity of the loaded code to a remote party would then be incorrect. Our model prevents this attack by assuming that code measured in PCRs during SRTM cannot be modified in memory.

**Comparison to SRTM.** The property provided by the DRTM protocol is stronger than the SRTM protocol for a number of reasons:

> **Fewer Assumptions.** The proof of security for the DRTM protocol does not rely on the $\mathsf{ProtectedSRTM}(m)$ assumption that static PCRs are locked. Instead the `latelaunch` action locks all dynamic PCRs. If the machine $M$ is a multi-processor or multi-core machine that is capable of running multiple threads in parallel, the locks on the dynamic PCRs will prevent attacks where malicious threads running concurrently with the measurement thread extend additional programs into $m.dpcr.k$ in an attempt to attest to their execution within a late launch session.
>
> **Smaller TCB.** The security proof of the DRTM does not reason about the measurements of the BIOS, boot loader, or operating system stored in the static PCRs (e.g., $m.pcr.s$), indicating that the security of the DRTM protocol does not depend on these large software components. This considerably reduces the trusted computing base to just $P(m)$ and $LL(m)$ and opens up the possibility of verifying that the TCB satisfies the required program invariants.
>
> **Execution Integrity.** Unlike the SRTM protocol that does not provide sufficient evidence to deduce that the last program in a sequence of measurements is branched to, the $J_{DRTM}$ property states that all programs measured during the protected session where executed. The property goes further to state that the programs completed execution. Specifically, the end of session measurement EOL proves that $P(m)$ executes to completion.

**Program Invariants.** In the process of proving the above theorem, we prove program invariants for the roles of the TPM (i.e., $TPM_{SRTM}(m)$ and $TPM_{DRTM}(m)$), and the programs $LL(m)$ and $P(m)$. These invariants specify the properties that $TPM_{SRTM}(m)$, $TPM_{DRTM}(m)$, $LL(m)$, and

$P(m)$ must satisfy for the DRTM protocol to be secure. The invariant over the roles of the TPM is similar to the TPM's role invariant used for SRTM. The invariant for $LL(m)$ states that the code must maintain a lock on $m.dpcr.k$ and measure then branch to the program $P(m)$. The invariant for $P(m)$ is shown below. The invariant states that if there are no resets or late launches on $m$ from $t_b$ to $t_e$, $m.dpcr.k$ is locked at $t_b$ and $m.dpcr.k$ contains the sequence $seq(dinit, P(m))$ at $t_b$ and later contains $seq(dinit, P(m), x, EOL))$, then there exists a thread $J$ such that $J$ extended a value $x$ (e.g., a nonce) into $m.dpcr.k$, then evaluated $f$, then extended the end of session symbol $EOL$, and that each action was performed once, in the order specified, and $m.dpcr.k$ was locked from $t_b$ to $t_X$.

$$
\begin{aligned}
[Q]_J^{t_b,t_e}\ \forall t,x.\ ((&\neg\mathsf{Reset}(m) \text{ on } (t_b,t_e])\\
&\wedge (\neg\mathsf{LateLaunch}(m) \text{ on } (t_b,t_e])\\
&\wedge (\mathsf{Mem}(m.dpcr.k, seq(dinit,P(m))) \text{ @ } t_b)\\
&\wedge (\mathsf{IsLocked}(m.dpcr.k,J) \text{ @ } t_b)\\
&\wedge (t_b < t \leq t_e)\\
&\wedge (\mathsf{Mem}(m.dpcr.k, seq(dinit,P(m),x,EOL)) \text{ @ } t))\\
&\qquad \supset\ \exists t_n,t_E,t_X.\ ((t_b < t_n < t_E < t_X < t)\\
&\qquad \wedge (\mathsf{Extend}(J,m.dpcr.k,x) \text{ @ } t_n)\\
&\qquad \wedge (\mathsf{Extend}(J,m.dpcr.k,EOL) \text{ @ } t_X)\\
&\qquad \wedge (\mathsf{Eval}(J,f) \text{ @ } t_E)\\
&\qquad \wedge (\neg\mathsf{Eval}(J,f) \text{ on } (t_b,t_E))\\
&\qquad \wedge (\neg\mathsf{Eval}(J,f) \text{ on } (t_E,t_X))\\
&\qquad \wedge (\mathsf{IsLocked}(m.dpcr.k,J) \text{ on } (t_b,t_X]))
\end{aligned}
$$

**Manual Audit of DRTM Implementation.** To check that the invariants required by our security analysis are correct, we performed a manual source code audit of the Flicker implementation of the DRTM protocol [30]. We checked that Flicker's security kernel implementation, represented by our program $P(m)$, respects the invariant above. We were able to quickly extract the security skeleton of the security kernel from Flicker's approximately 250 lines of C code. To verify that the skeleton respects the exact invariant from our security proof, we checked that instructions were present to evaluate the function $f$, that the $EOL$ marker was subsequently extended into $m.dpcr.k$, and that each of the instructions would only be executed once on all code paths. In several cases, we matched multiple C instructions to a single action since the instructions are a refinement of the action. For example, the extension of $EOL$ consists of two instructions, a `memset` to create the sequence of characters corresponding to an $EOL$ and a call to a wrapper for the extend instruction. The entire manual process of extracting the security skeleton and auditing the invariant took less than one hour for an individual with no previous experience with the Flicker security kernel. Although we did not formally verify the property, one interesting direction for future work is to use these invariants to derive refined invariants to check on the implementation, possibly using software model checking techniques.

## 5. Related Work

$LS^2$ draws on certain conceptual ideas from PCL [3], in particular, the local reasoning style by which security properties of protocols are proved without explicitly reasoning about adversary actions. In PCL, global security properties are derived by combining properties achieved by individual protocol steps with invariants proved by induction over the protocol programs executed by honest parties. $LS^2$ supports this form of reasoning for a much richer language that includes not only network communication and cryptography as in PCL, but also shared memory, memory protection, machine resets, and dynamically loaded unknown pieces of code. The insights on which the new proof rules are based are described in Section 2.2. The technical definition of $LS^2$ also differs significantly from PCL: instead of associating pre-conditions and post-conditions with all actions in a process (as PCL does), we model time explicitly, and associate monotonically increasing time points with events on a trace. The presence of explicit time allows us to express invariants about memory; for instance, we may express in $LS^2$ that a memory location contains the same value throughout the interval $[t_1, t_2]$. Explicit time is also used to reason about the relative order of events. Whereas explicit use of time may appear to be low-level and cumbersome for practical use, the proof system for $LS^2$ actually uses time in a very limited way that is quite close to temporal logics such as LTL [31]. Indeed, it seems plausible to rework the proof system in this paper using operators of LTL in place of explicit time. However, we refrain from doing so because we believe that a model of real time may be needed to analyze some systems of interest (e.g., [32]–[34]).

$LS^2$ also shares some features with other logics of programs [8], [10], [35]. Hoare logic and dynamic logic focus on sequential imperative programs, and do not consider concurrency, network communication and adversaries. $LS^2$'s abstract locks are similar to regions that are used to reason about synchronized access to memory in concurrent separation logic [8]. However, the two primitives differ in application. Whereas we use locks to enforce integrity of data stored in memory, regions are intended to prevent race conditions. Another key difference between concurrent separation logic and $LS^2$ is that the former does not consider network communication. Furthermore, concurrent separation logic and other approaches for verifying concurrent systems [36] 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.

Prior proposals for reasoning about dynamically loaded code use higher-order extensions of Hoare logic [24]–[28]. However, they are restricted to reasoning about sequential programs only and require that invariants of code being called be known in the program at the point of the call. $LS^2$'s method addresses the problem of reasoning about dynamically loaded code in the more general context of concurrent program execution where one thread is allowed to modify code that is loaded by another. As illustrated in Section 4.1, using the (Jump) rule, evidence that *some* code executed can be combined with separate evidence about the identity of the code to reason precisely about the effects of the jump. Such reasoning is essential in some applications including trusted computing, and is impossible in all prior work known to us.

There have been several previous analyses of trusted computing. Abadi and Wobber used an authorization logic to describe the basic ideas of NGSCB, the predecessor to the TCG [37]. Their formalization documents and clarifies basic NGSCB concepts rather than proving specific properties of systems utilizing a TPM. Chen et al. developed a formal logic tailored to the analysis of a remote attestation protocol and suggested improvements [38]. Unlike $LS^2$, these logics are not tied to the execution semantics of the protocols. Gurgens et al. used a model checker to analyze the security of several TCG protocols [39]. Millen et al. employed a model checker to understand the role and trust relationships of a system performing a remote attestation protocol [40]. Our analysis with $LS^2$ is a complementary approach: It proves security properties even for an infinite number of simultaneous invocations of attestation protocols, but with a more abstract model of the TPM's primitives. $LS^2$ is designed to be a more general logic with TCG protocols providing one set of applications. Lin [41] used a theorem prover and model finder to analyze the security of the TPM against invalid sequences of API calls.

## 6. Conclusion

In this paper, we presented $LS^2$ and used it to carry out a substantial case study of trusted computing attestation protocols. The design of $LS^2$ was conceptually and technically challenging. Specifically, it was difficult to define a realistic adversary model and formulate sound reasoning principles for dynamically loaded unknown (and untrusted) code. The proof system was designed to support reasoning at a high level of abstraction. This was particularly useful in the case studies where the proofs yielded many insights about the security of trusted computing systems.

In future work, we will build upon this work to model and analyze security properties of web browsers, security hypervisors and virtual machine monitors. We also plan to develop further principles for modeling and reasoning about security at the level of system interfaces, in particular, to support richer access control models and system composition and refinement.

**Acknowledgments.** The authors would like to thank Michael Hicks, Jonathan McCune, and the anonymous reviewers for their helpful comments and suggestions. This work was partially supported by the U.S. Army Research Office contract on Perpetually Available and Secure Information Systems (DAAD19-02-1-0389) to CMU's CyLab, the NSF Science and Technology Center TRUST, and the NSF CyberTrust grant "Realizing Verifiable Security Properties on Untrusted Computing Platforms". Jason Franklin is supported in part by an NSF Graduate Research Fellowship.

## References

[1] "Trusted Computing Group (TCG)," https://www.trustedcomputinggroup.org/, 2008.

[2] A. Datta, A. Derek, J. C. Mitchell, and D. Pavlovic, "A derivation system and compositional logic for security protocols," *Journal of Computer Security*, vol. 13, no. 3, pp. 423–482, 2005.

[3] A. Datta, A. Derek, J. C. Mitchell, and A. Roy, "Protocol Composition Logic (PCL)." *Electr. Notes Theor. Comput. Sci.*, vol. 172, pp. 311–358, 2007.

[4] N. Durgin, J. C. Mitchell, and D. Pavlovic, "A compositional logic for proving security properties of protocols," *Journal of Computer Security*, vol. 11, pp. 677–721, 2003.

[5] A. Roy, A. Datta, A. Derek, J. C. Mitchell, and J.-P. Seifert, "Secrecy analysis in protocol composition logic," *Formal Logical Methods for System Security and Correctness*, 2008.

[6] R. Milner, M. Tofte, and R. Harper, *The Definition of Standard ML*. Cambridge, MA, USA: MIT Press, 1990.

[7] J. Saltzer and M. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, September 1975.

[8] S. Brookes, "A semantics for concurrent separation logic," in *Proceedings of 15th International Conference on Concurrency Theory*, 2004.

[9] Z. Manna and A. Pnueli, *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1995.

[10] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[11] P. W. O'Hearn, J. C. Reynolds, and H. Yang, "Local reasoning about programs that alter data structures," in *Proceedings of the 15th International Workshop on Computer Science Logic*. London, UK: Springer-Verlag, 2001, pp. 1–19.

[12] J. C. Reynolds, “Separation logic: A logic for shared mutable data structures,” in *Proceeding of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society, 2002, pp. 55–74.

[13] Trusted Computing Group, “TCG Specification Architecture Overview, Specification Revision 1.4,” https://www.trustedcomputinggroup.org/groups/TCG_1_4_Architecture_Overview.pdf, August 2007.

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

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

[16] “Intel Trusted Execution Technology: Software Development Guide,” Document Number: 315168-005, Intel Corporation, June 2008.

[17] A. Datta, J. Franklin, D. Garg, and D. Kaynar, “A logic of secure systems and its application to trusted computing,” Carnegie Mellon University, Tech. Rep. CMU-CyLab-09-001, 2009.

[18] E. M. Chan, J. C. Carlyle, F. M. David, R. Farivar, and R. H. Campbell, “BootJacker: Compromising computers using forced restarts,” in *Proceedings of 15th ACM Conference on Computer and Communications Security*, 2008.

[19] S. Garriss, R. Cáceres, S. Berger, R. Sailer, L. van Doorn, and X. Zhang, “Towards trustworthy kiosk computing,” in *Workshop on Mobile Computing Systems and Applications*, Feb. 2006.

[20] H. DeYoung, D. Garg, and F. Pfenning, “An authorization logic with explicit time,” in *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF-21)*, Jun. 2008.

[21] J. Reed, “Hybridizing a logical framework,” in *International Workshop on Hybrid Logic 2006 (HyLo 2006)*, ser. Electronic Notes in Computer Science, August 2006.

[22] T. Braüner and V. de Paiva, “Towards constructive hybrid logic,” in *Electronic Proceedings of Methods for Modalities 3 (M4M3)*, 2003.

[23] “Secure virtual machine architecture reference manual.” AMD Corp., May 2005. [Online]. Available: http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/33047.pdf

[24] N. Krishnaswami, “Separation logic for a higher-order typed language,” 2006, in Workshop on Semantics, Program Analysis and Computing Environments for Memory Management, SPACE06.

[25] H. Thielecke, “Frame rules from answer types for code pointers,” in *33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: ACM, 2006, pp. 309–319.

[26] Z. Ni and Z. Shao, “Certified assembly programming with embedded code pointers,” in *33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: ACM, 2006, pp. 320–333.

[27] H. Cai, Z. Shao, and A. Vaynberg, “Certified self-modifying code,” in *ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, 2007, pp. 66–77.

[28] A. Nanevski, G. Morrisett, and L. Birkedal, “Hoare type theory, polymorphism and separation,” *Journal of Functional Programming*, vol. 18, no. 5&6, pp. 865–911, 2008.

[29] B. Kauer, “OSLO: Improving the security of trusted computing,” in *Proceedings of the USENIX Security Symposium*, Aug. 2007.

[30] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, “Flicker: An execution infrastructure for tcb minimization,” in *Proceedings of the ACM European Conference in Computer Systems (EuroSys)*, Apr. 2008.

[31] A. Pnueli, “The temporal logic of programs,” in *Proceedings of 19th Annual Symposium on Foundations on Computer Science*, 1977.

[32] R. Kennell and L. H. Jamieson, “Establishing the genuinity of remote computer systems,” in *Proceedings of the 2003 USENIX Security Symposium*, Aug. 2003.

[33] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla, “SWATT: Software-based attestation for embedded devices,” in *Proceedings of the IEEE Symposium on Security and Privacy*, May 2004.

[34] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla, “Pioneer: Verifying code integrity and enforcing untampered code execution on legacy platforms,” in *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2005.

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

[36] L. Lamport, “The temporal logic of actions,” *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 3, May 1994.

[37] M. Abadi and T. Wobber, “A logical account of NGSCB,” in *Proceedings of Formal Techniques for Networked and Distributed Systems*, 2004.

[38] S. Chen, Y. Wen, and H. Zhao, “Formal analysis of secure bootstrap in trusted computing,” in *Proceedings of 4th International Conference on Autonomic and Trusted Computing*, 2007.

[39] S. Gurgens, C. Rudolph, D. Scheuermann, M. Atts, and R. Plaga, “Security evaluation of scenarios based on the TCG’s TPM specification,” in *Proceedings of 12th European Symposium On Research In Computer Security*, 2007.

[40] J. Millen, J. Guttman, J. Ramsdell, J. Sheehy, and B. Sniffen, “Analysis of a measured launch,” The MITRE Corporation, Tech. Rep., 2007.

[41] A. H. Lin, “Automated analysis of security apis,” Master’s thesis, Massachusetts Institute of Technology, 2005.