Towards a Theory of Secure Systems

Deepak Garg Jason Franklin Dilsun Kaynar Anupam Datta

February 4, 2008
CMU-CyLab-08-003

CyLab
Carnegie Mellon University
Pittsburgh, PA 15213

Towards a Theory of Secure Systems

Deepak Garg Jason Franklin Dilsun Kaynar Anupam Datta
Carnegie Mellon University

Abstract

We initiate a program to develop a principled theory of secure systems. Our main technical result is a formal logic for reasoning about a network of shared memory, multi-user systems. The logic is inspired by an existing logic for security protocols. It extends the attacker model and adds shared memory, time, and limited forms of access control. We prove soundness for the proof system in the presence of an attacker who controls the network and has partial control over shared memory on individual machines. We illustrate the use of the logic by proving a relevant security property of a part of the Trusted Computing Group's remote attestation protocol.

1. Introduction

This paper initiates a program to develop a theoretical basis for the design and analysis of secure systems. For the purpose of this paper, we define a *system* as a set of concurrent, interacting threads with shared mutable state. Interaction may take place locally through reads and writes to shared memory or over a network. This definition is motivated by our target application domain, which includes a number of deployed and industrial-standard contemporary systems such as OS kernels, virtual machine monitors (VMMs), and co-processor-based systems such as those utilizing the Trusted Computing Group's Trusted Platform Module (TPM) [31].

Our long-term goal is to develop a formal framework for modeling and analysis of secure systems at two levels of abstraction—*system architecture* and *system implementation*. A specific issue that we plan to address in developing and using this framework is to provide rigorous *definitions of security* and *adversary models*, a relatively unexplored area in system security. In addition, we hope to identify *design principles* for secure systems, as well as a core set of basic *building blocks* from which complex systems can be constructed via secure composition.

In this paper, we take several steps towards achieving this goal. We introduce the *Logic of Secure Systems* (LS^2), a logic for reasoning about security properties of systems at the architecture level. The logic is designed around a programming language with two distinctive features. First, the language includes standard process calculus style communication primitives as well as imperative constructs for reading and updating memory. This (somewhat non-standard) design choice is motivated by our goal to (a) have a language in which the representation of the system is close to its actual implementation, and (b) to develop simple, high-level reasoning principles for such systems. Using the replication operator from process calculus would, while providing a way to encode state, defeat these goals. Second, the language includes primitives for modeling *memory protection*, a fundamental building block for secure systems [27], at a high-level of abstraction. The operational semantics defines the set of traces produced by a program. Adversaries are modeled as programs in the same language. The adversary controls the complete network and can write to any unprotected memory location. Traces are augmented by associating time with events. We use a dense total order to model time (i.e. between any two time points, there exists another) and require that successive events on a trace have monotonically increasing time values associated with them. The extension to a timed model is motivated by the need to specify meaningful properties of secure systems. While a temporal logic such as LTL [25] may suffice for certain applications, we expect that a model of real time may be needed to analyze some systems of interest [21, 28, 29].

LS^2 draws its lineage from a logic for network protocol analysis, *Protocol Composition Logic (PCL)* [4–13, 16, 26]. It extends PCL by incorporating time in all formulas, and by introducing new axioms for reasoning about shared memory, memory protection and machine resets. It retains other features of PCL, such as predicates for reasoning about signatures and their verifications. A major departure from PCL is the basic style of reasoning: PCL associates pre-conditions and post-

conditions with each action in a process; the presence of time in LS^2 allows us to reason globally about the program (the order of events is captured by the natural order on time). This difference results in significant differences in the judgments of the two logics. The central judgment of the logic is $\vec{l}, t_b, t_e \vdash [P]_X\ A$, which intuitively means that if X executes the program P between time t_b and t_e while starting from a configuration that locks all locations in $\vec{l}$ for X (so that only X has write access to these locations), then the formula A holds. The proof system for LS^2 allows us to derive such judgments. It consists of about 20 generic rules for reasoning about secure systems and 10 application-specific rules for reasoning about trusted computing systems. We prove a *soundness theorem* for the proof system. We illustrate the use of the logic by proving a relevant security property of a part of the Trusted Computing Group's remote attestation protocol. Our proof indicates in a precise way certain security issues with the protocol (see Section 5.2 for details).

The rest of the paper is organized as follows. Section 2 presents an example of a trusted computing system protocol that we use as a running example throughout the paper. The full definition of the syntax and operational semantics of the programming language as well as the timed trace model is given in Section 3. Section 4 describes the syntax and semantics of LS^2. Section 5 presents a proof system for LS^2, its soundness theorem and the application to the trusted computing protocol. Conclusions appear in Section 7. Detailed proofs are in the appendices.

2. Motivating Example

As an illustration of our logic and proof method, we prove the correctness of a simplified version of the Trusted Computing Group's (TCG's) remote attestation protocol. The protocol runs between a platform and a remote verifier. The objective of the protocol is to convince the remote verifier of the integrity of the platform's boot sequence.

Trusted Platform Module. To achieve this goal, a platform utilizes a secure co-processor called the Trusted Platform Module (TPM). Conceptually, the TPM provides cryptographic primitives, shielded locations, protected capabilities, and a starting point for the measurement process called the Core Root of Trust for Measurement (CRTM). In today's systems, the CRTM may be in the BIOS rather than in the TPM, hence we use CRTM and BIOS interchangeably. There are two versions of the attestation protocol which differ based on the point at which measurement begins. In this paper, we focus on the standard Static Root of Trust for Measurement (SRTM) protocol which begins measurement at system boot. This protocol relies on the TCG's Platform Configuration Registers (PCRs) and `TPM_Extend` command. PCRs are protected registers that can only be written by two special instructions, one for resetting and one for extending stored values. We denote the PCRs of a machine m by $m.PCR(i)$ where i ranges over natural numbers. The instruction `reset` m puts the special value 0 in all the PCRs of machine m. It is assumed that this instruction is executed only when a machine is rebooted, and that a PCR cannot obtain this special value in any other way. The instruction `extend` $m.PCR(k),\ m.loc$ modifies the content of $m.PCR(k)$ to be a hash of the concatenation of its current value with a hash of the contents of the memory location $m.loc$ ($m.loc$ denotes an abstract memory location loc on machine m). In pseudocode, the effect of this instruction may be summarized as follows:

$$m.PCR(k) \leftarrow HASH(m.PCR(k)||HASH(m.loc))$$

We often use the notation $seq(0, v_1, \ldots, v_n)$ ($n \geq 0$) to denote the contents of a PCR which started with 0 and has subsequently been extended with the sequence values $v_1, \ldots, v_n$. Formally, $seq(0) = 0$ and $seq(0, v_1 \ldots, v_n, v_{n+1}) = HASH(seq(0, v_1, \ldots, v_n)||HASH(v_{n+1}))$.

A Modified SRTM Protocol. As an illustrative example, we describe a simplified version of the SRTM attestation protocol in Figure 1. Subsequent sections describe an analysis of this protocol using the proof system of LS^2.

The SRTM protocol is a three agent protocol, each agent executing a different process. The three processes are: (1) $Booting(m)$, executed by the machine itself (called $\hat{m}$), that measures the boot loader and operating system, and communicates with the remote verifier, (2) $Signer(m)$, executed by the TPM of m, that signs the PCR containing code measurements, and (3) $Verifier(m)$, executed by a remote verifier who wants to verify the integrity of m's boot sequence.

The correctness of the SRTM protocol relies on a number of non-trivial facts. First, it depends on the fact that a PCR can only be extended, and never overwritten in any other way, and hence always contains a record of everything that was written to it since the last reboot. Second, this protocol (implicitly) depends on write-protection in memory. For instance, it is critical

$Booting(m)$	$\equiv$	`extend` $m.PCR(s), m.bl\_loc$;	Extend bootloader's code into $PCR(s)$
		$b =$ `read` $m.bl\_loc$;	Read bootloader's code
		`call` b;	Call the bootloader
		`extend` $m.PCR(s), m.os\_loc$;	Extend operating system's code into $PCR(s)$
		$o =$ `read` $m.os\_loc$;	Read the operating system
		`call` o;	Call the operating system
		`unlock` $locs(m)$;	Unlock all locations
		...	
		`send` ();	Signal $Signer(m)$ to sign $m.PCR(s)$
		$s =$ `receive` ;	Receive signature from $Signer(m)$
		`send` s	Send signature to $Verifier(m)$
$Signer(m)$	$\equiv$	_ = `receive` ;	Receive signal from $Booting(m)$
		$w =$ `read` $m.PCR(s)$;	Read $PCR(s)$
		$s =$ `sign` w;	Sign the value in $PCR(s)$
		`send` s	Send signature to $Booting(m)$
$Verifier(m)$	$\equiv$	$s =$ `receive` ;	Receive signed $PCR(s)$ from $Booting(m)$
		`verify` $s, seq(0, BL(m), OS(m)), \hat{TPM}(m)$	Verify the signature

Figure 1. A Modified SRTM Protocol

that the code of the boot loader remain the same between the time it is measured into a PCR by and invoked. Proving the protocol correct, therefore, requires a formalism that can deal with both PCRs and shared memory with write locks.

We have simplified the protocol significantly from its actual specification. For example, in our presentation, both the boot loader and the operating system are loaded by the BIOS (as opposed to the actual protocol in which the operating system is loaded by the boot loader). As a result, we do not have to model the exact code of the boot loader, and also avoid having to model function calls in our formalism. While we believe that our logic can be extended to handle such branching code, we have not worked out the extension in full detail yet.

Other simplifications we make are: (1) We do not model software loaded after the operating system, and assume that only one PCR is used for all hashes. Adding further layers of software, or using more PCRs does not change the structure of the proof of correctness, it merely repeats steps. (2) We ignore a nonce that is intended to avoid replay attacks in the actual protocol. This change weakens the guarantee that is available to verifier. We can easily model this nonce in our formalism, but its addition does not highlight any new technique or method. (3) We do not model the transmission or verification of a certificate or a measurement list containing the list of programs whose hashes are included in the PCRs. We also assume that the TPM is genuine and that the measured programs are known a priori.

3 Modeling Secure Systems

In this section, we present the syntax and operational semantics for our programming language. In addition, we define precisely how time is added to the traces obtained by executing programs.

3.1 Process calculus

Figure 2 summarizes the syntax of our process calculus. We assume a fixed set $\mathcal{M}$ of machines (denoted m), and a fixed set $\mathcal{L}$ of memory locations spanning all machines (denoted l). We write $machine(l)$ to denote the machine on which l is located. Often, we explicitly qualify a location with the name of the machine on which it is located by writing $m.l$. We use the notation $\vec{l}$ to denote a *set* of locations, and $locs(m)$ to denote the set of all locations on machine m.

There is a special set of locations on each machine m denoted $m.PCR(x)$, representing the platform configuration registers (PCRs) on machine m (x is the index of the PCR). PCRs cannot be read or written in the usual way; instead they can only be extended or reset.

Each machine may execute a number of concurrent threads. There are send and receive primitives that allow message based communication between threads. However, communication is undirected and insecure — any message sent by any thread may be received by any other thread, even perhaps on a remote machine.

Machine	m			
Location	l			
Agent/Public Key	$\hat{K},\hat{X},\hat{Y}$			
Unique thread identifier	η			
Thread id	X,Y	::=	$\langle \hat{X},\eta,\mathrm{m}\rangle$	
Value/expression	e,v	::=	n	Number
		|	$\hat{K}$	Agent/Public Key
		|	$v\|\|v'$	Pair
		|	$HASH(v)$	Hash of v
		|	$SIG_{\hat{K}}\{\|v\|\}$	Value v signed by $\hat{K}$'s private key
		|	c	Code that can be called
		|	x	Variable
Action	a	::=	`extend` $PCR(x),l$	Extend $PCR(x)$ with value at location l
		|	`read` l	Read location l
		|	`write` l,v	Write v to location l
		|	`call` c	Call the code c
		|	`send` v	Send v as a message
		|	`receive`	Receive a message
		|	`sign` v	Sign v with private key
		|	`verify` s,v,K	Verify that $s = SIG_{\hat{K}}\{\|v\|\}$
		|	`lock` l	Obtain write lock on location l
		|	`unlock` l	Release write lock on location l
Process	P,Q	::=	$x_1 = a_1;\ldots;x_n = a_n$	
Thread	T,S	::=	$\langle X,P\rangle$	
Store (function)	σ	:	$\mathcal{L} \rightarrow$ Values	
Lock map (function)	ζ	:	$\mathcal{L} \rightarrow$ (Thread ids) $\cup$ $\{\_\}$	
Configuration	C	::=	$\zeta,\sigma \vdash T_1\|\ldots\|T_n$	

Figure 2. Syntax of the process calculus

We also assume a set of agents $\hat{X}$, $\hat{Y}$, $\hat{K}$, etc. that execute programs, possibly many at one time on several machines. Each agent has a unique public or verification key, which is identified with the name of the agent. The private or signing key of an agent $\hat{X}$ is represented by the notation $\hat{X}^{-1}$. Some agents may participate in a given protocol, others may be intruders and yet others may be non-interfering observers. All these agents are modeled the same way. We stipulate for each machine m a special agent called $\hat{m}$, which owns any threads running on behalf of the machine itself.

Each thread is identified by a unique id, denoted X. The id of a thread is actually a 3-tuple $\langle \hat{X},\eta,m\rangle$ where $\hat{X}$ is the name of the agent on behalf of whom the thread runs, η is a unique thread identifier (used to distinguish multiple threads of the same agent), and m is the machine on which the thread runs. We define $machine(X) = m$ if $X = \langle \hat{X},\eta,m\rangle$.

The program of a thread is called a *process* (P, Q, etc.). Processes are written in functional style, with variable names denoting expressions and values (denoted e and v, respectively), and reduction rules substituting values for variables. We implicitly assume that values and expressions are typed. The types include numbers, keys, pairs, signatures, hashes, and code. For the purposes of this paper, we do not need to distinguish between values and expressions.

A salient point is that machine names and locations are not expressions. In particular variables can't range over them, nor can they be passed in messages or stored in locations. This restriction is necessary to be able to accurately track machine names and locations in our proof system.

Formally, processes are (possibly empty) sequences of actions. Each *action* (a) performs either communication, or changes the state of memory (on the local machine), or performs some evaluation such as signature computation or signature verification. All actions return a result, which immediately binds to a variable (written x_i in the figure). The bound variable x_i is in scope in the part of the process following the action. We freely allow renaming of bound variables.

For an action such as reading a memory location or receiving a message, the value returned is the obvious one obtained by performing the action. For actions such as writing to memory or sending a message, the value

returned is a dummy (one may assume it to be zero). In the latter case, we often omit the variable bound by the action, writing _ in its place. The special action `call` c describes a function call to the function whose code is c. In our calculus, this call does nothing, except recording a reduction on the trace, which can then be reasoned about in the logic. The actions `lock` l and `unlock` l acquire and release a write-lock on location l.

A *thread*, denoted S, T, is formally a pair $\langle X, P\rangle$, where X and P represent the id and the process of the thread, respectively.

A snapshot of all threads running on all machines at a point of time is called a *configuration*, denoted C. In addition to threads, a configuration contains two other pieces of information: (1) A *store*, σ that maps each location in $\mathcal{L}$ to the value stored in it, and (2) A *lock map*, ζ that maps each location in $\mathcal{L}$ to the id of the thread that has a lock on it. If no thread has a lock on a location, the location is mapped to the special symbol _. The locks we consider in this paper are write-locks; a location locked by a thread may still be read by other threads, but it may be written only by the thread that holds the lock.

(It is relatively straightforward to conceive a different process calculus with locks that restrict both writing and reading. However, we do not consider such locks in this paper.)

For any thread X, we define $locked(X, \zeta)$ as the subset of $\mathcal{L}$ that contains exactly the locations which are locked by X according to ζ, i.e, $locked(X, \zeta) = \{l \in \mathcal{L} | \zeta(l) = X\}$. Similarly, we define $unlocked(\zeta) = \{l \in \mathcal{L} | \zeta(l) = \_\}$. In reality, both the store and the lock map would be maintained on a machine by machine basis. Our notation is equivalent to this real situation, since the machine associated with any location can be obtained simply by examining it.

Well-formed configurations. Not all configurations are well-formed. For well-formedness, we require the following four syntactic conditions. Let $mustlocked(P)$ be the set of locations that occur in P in actions of the form `unlock` l, that are not preceded by a corresponding `lock` l. Intuitively, these are the locations that must be locked before P is started.

1. The processes of all threads are closed, i.e., without free variables.

2. No thread in the configuration should try to lock a location that it already has a lock on. This property can be checked syntactically by examining the process of the thread since locations cannot be bound as variables.

3. For any thread $\langle X, P\rangle$ in the configuration, $mustlocked(P) \subseteq locked(X, \zeta)$. This ensures that an unlock action always succeeds.

4. The process P of each thread $\langle X, P\rangle$ mentions locations contained in $machine(X)$ only. This means that remote locations cannot be accessed by threads.

In the rest of the paper, we assume that all configurations are well-formed, without explicit reference to this fact. The reader may check that our reduction rules preserve well-formedness of configurations.

Honesty. We call an agent $\hat{X}$ honest, if the agent does not leak its private (signing) key. In our logic, we write this as the predicate $\mathsf{Honest}(\hat{X})$. We assume that for each agent it is known whether the agent is honest or not, and that this fact remains unchanged as the configuration evolves. For an honest agent, we may optionally specify the possible processes $\vec{P}$ ($\vec{P}$ denotes a set of processes) that threads owned by it may execute. In our logic this is written using the binary predicate $\mathsf{Honest}(\hat{X}, \vec{P})$. Such information is often useful for reasoning about the actions of the process.

Reduction Rules. The reduction rules for configurations are shown in Figure 3. We use the notation $\sigma[l \mapsto v]$ to mean the store σ, with the change that l contains the value v. Similarly, $\zeta[l \mapsto X]$ means the map ζ, with the exception that l is locked by X (if $X = \_$, then l is unlocked).

The reduction rules (with the exception of (reset)) are associated with actions in processes. Each action corresponds to one rule. Rules (extend) to (unlock) capture internal reductions within a thread.

The rule (comm) captures synchronous communication between two threads, one willing to send a value and the other willing to receive a value. There is no restriction on communication; arbitrary threads may synchronize with each other.

The rule (reset) represents the action of resetting a machine m. An important point is that this rule does not correspond to any action in the process calculus; we assume that any machine can be reset at any time spontaneously. Resetting a machine kills some threads that are on the machine. The reason for not killing all threads is that some threads which are modeled in the configuration at the time of reset may not actually have started by then, and may be waiting to start later. We stipulate that any threads holding locks be necessarily killed.

The notation $(T_1 \mid \ldots \mid T_n) - \{m\}$ denotes an arbitrary subset of threads in $T_1 \mid \ldots \mid T_n$, the only removed

$$
\begin{array}{ll}
\text{(extend)} & \zeta, \sigma[m.PCR(x) \mapsto seq(0, v_1, \ldots, v_n)][l \mapsto v_{n+1}] \vdash \langle X, \_ = \texttt{extend}\ m.PCR(x), l;\ P\rangle \mid \ldots \\
& \quad \longrightarrow\ \zeta, \sigma[m.PCR(x) \mapsto seq(0, v_1, \ldots, v_n, v_{n+1})][l \mapsto v_{n+1}] \vdash \langle X, P\rangle \mid \ldots \\
& (\text{where } machine(X) = m \text{ and } m.PCR(x) \in locked(X, \zeta) \cup unlocked(\zeta)) \\
\text{(read)} & \zeta, \sigma[l \mapsto v] \vdash \langle X, x = \texttt{read}\ l;\ P\rangle \mid \ldots \ \longrightarrow\ \zeta, \sigma[l \mapsto v] \vdash \langle X, P[v/x]\rangle \mid \ldots \\
\text{(write)} & \zeta, \sigma[l \mapsto v'] \vdash \langle X, \_ = \texttt{write}\ l, v; P\rangle \mid \ldots \ \longrightarrow\ \zeta, \sigma[l \mapsto v] \vdash \langle X, P\rangle \mid \ldots \\
& (\text{where } machine(X) = m \text{ and } l \in locked(X, \zeta) \cup unlocked(\zeta)) \\
\text{(call)} & \zeta, \sigma \vdash \langle X, \_ = \texttt{call}\ c; P\rangle \mid \ldots \ \longrightarrow\ \zeta, \sigma \vdash \langle X, P\rangle \mid \ldots \\
\text{(sign)} & \zeta, \sigma \vdash \langle X, x = \texttt{sign}\ v; P\rangle \mid \ldots \ \longrightarrow\ \zeta, \sigma \vdash \langle X, P[SIG_{\hat{X}}\{|v|\}/x]\rangle \mid \ldots \\
\text{(verify)} & \zeta, \sigma \vdash \langle X, \_ = \texttt{verify}\ s, v, K; P\rangle \mid \ldots \ \longrightarrow\ \zeta, \sigma \vdash \langle X, P\rangle \mid \ldots \\
& (\text{if } s = SIG_{\hat{K}}\{|v|\}) \\
\text{(lock)} & \zeta[l \mapsto \_], \sigma \vdash \langle X, \_ = \texttt{lock}\ l; P\rangle \mid \ldots \ \longrightarrow\ \zeta[l \mapsto X], \sigma \vdash \langle X, P\rangle \mid \ldots \\
\text{(unlock)} & \zeta[l \mapsto X], \sigma \vdash \langle X, \_ = \texttt{unlock}\ l; P\rangle \mid \ldots \ \longrightarrow\ \zeta[l \mapsto \_], \sigma \vdash \langle X, P\rangle \mid \ldots \\
\text{(comm)} & \zeta, \sigma \vdash \langle X, \_ = \texttt{send}\ v; P\rangle \mid \langle Y, x = \texttt{receive}\ ; Q\rangle \mid \ldots \\
& \quad \longrightarrow\ \zeta, \sigma \vdash \langle X, P\rangle \mid \langle Y, Q[v/x]\rangle \mid \ldots \\
\text{(reset)} & \zeta, \sigma \vdash T_1 \mid \ldots \mid T_n \\
& \quad \longrightarrow\ \zeta[locs(m) \mapsto X], \sigma[locs(m) \mapsto 0] \vdash (T_1 \mid \ldots \mid T_n) - \{m\} \mid \langle X, Booting(m)\rangle \\
& (\text{where } X = \langle \hat{m}, \eta, m\rangle \text{ and } \eta \text{ is fresh})
\end{array}
$$

Figure 3. Reduction Rules of the Process Calculus

threads being situated on machine m. A special thread, X, owned by $\hat{m}$ is started on m after a reset. This thread runs a fixed process $Booting(m)$, which represents the actions needed for booting the machine. The exact details of this process depend on the way machines are modeled (which we keep abstract). For the illustrative example that we consider in this paper, this process is defined in Figure 1. In addition, resetting machine m acquires locks on all locations on the machine for the new thread X (captured by the notation $\zeta[locs(m) \mapsto X]$) and sets all locations on the machine, including PCRs, to 0 (captured by the notation $\sigma[locs(m) \mapsto 0]$).

Initial configurations. A configuration is called initial if it satisfies the following conditions:

1. No value in the range of the store contains a subexpression of the form $SIG_{\hat{X}}\{|v|\}$, if $\hat{X}$ is honest.
2. For each agent $\hat{X}$, no thread of agent $\hat{X}$ contains a value of the form $SIG_{\hat{Y}}\{|v|\}$ if $\hat{X} \neq \hat{Y}$ and $\hat{Y}$ is honest.
3. Each PCR, on every machine, contains a value other than 0 that cannot be written as $HASH(v)$ for any v.

Conditions (1) and (2) state that in an initial configuration, messages signed by honest agents should neither be known to other agents, nor stored in memory. Together they imply that, starting from an initial configuration with honest $\hat{X}$, a process of a different agent $\hat{Y}$ may contain an expression of the form $SIG_{\hat{X}}\{|v|\}$ only if $\hat{X}$ actually sent it in a message. This is necessary to prove the correctness of some protocols. Condition (3) is a technical condition needed to prove soundness. One may assume that all PCRs initially contain the value 1.

Adversary Model. Adversaries are modeled as additional threads in a configuration. There is no constraint on adversaries, except that they must respect conditions (1) and (4) of well-formed configurations and condition (2) of initial configurations. In particular, adversaries do not have to be honest, and they may intercept any communication, read any memory location, and write any unlocked location. An important point to observe here is that, owing to a restricted syntax of expressions, adversaries are limited in what they can do with intercepted messages. For example, there is no construct for projecting the components of a pair. However, the restricted syntax of expressions is merely a matter of presentation; there is no new technical difficulty in working with a bigger syntax of expressions.

3.2 Timed Traces and Matching

A *trace* is a sequence of configurations $C_0, \ldots, C_n$, such that (1) C_0 is an initial configuration, and (2) for each i, C_i reduces to C_{i+1} using one of the reduction rules of the process calculus.

A *timed trace* (denoted $\mathcal{T}$) is a trace, in which a time point (real number) is associated with each reduction

step. This time point represents the time at which the reduction occurs. We require that the time points be monotonically increasing. We often write a timed trace as ($t_1 < \ldots < t_n$ are the reduction time points):

$$C_0 \xrightarrow{t_1} C_1 \ldots \xrightarrow{t_n} C_n$$

Let θ, φ denote substitutions mapping program variables to ground expressions, and $P\theta$ denote the result of applying θ to P. We say that a timed trace $\mathcal{T}$ matches the tuple $\langle \vec{l}, t_b, t_e, X, P \rangle$ with substitution θ if the reductions of X's process in some subtrace of $\mathcal{T}$ match the sequence of actions in $P\theta$, all these reductions happen during the semi-open interval $[t_b, t_e)$, and at the start of the first action in P, each location in $\vec{l}$ is locked for X. The reason for using semi-closed intervals such as $[t_b, t_e)$ instead of open or closed intervals is mainly technical convenience.

Formally, we say that $\mathcal{T} \gg \langle \vec{l}, t_b, t_e, X, P \rangle \mid \theta$ (read "$\mathcal{T}$ matches $\langle \vec{l}, t_b, t_e, X, P \rangle$ with substitution θ") if the following hold:

1. For some $C_i \in \mathcal{T}$ and process Q, $C_i = \zeta_i, \sigma_i \vdash \langle X, (P;Q)\theta \rangle \mid \ldots$
2. $\mathit{locked}(X, \zeta_i) \supseteq \vec{l}$
3. For some $i' \geq i$, $C_{i'} \in \mathcal{T}$, and substitution φ, $C_{i'} = \zeta_{i'}, \sigma_{i'} \vdash \langle X, (Q\theta)\varphi \rangle \mid \ldots$
4. The time associated with each reduction of X between C_i and $C_{i'}$ lies in the interval $[t_b, t_e)$.

If P does not contain any actions, we trivially have $\mathcal{T} \gg \langle \vec{l}, t_b, t_e, X, P \rangle \mid \theta$ for any substitution θ, if in $\mathcal{T}$, X does not perform any reduction in the interval $[t_b, t_e)$.

For an action a, we define $\mathcal{T} \gg \langle \vec{l}, t_b, t_e, X, y, a \rangle \mid \theta, v/y$ if $\mathcal{T} \gg \langle \vec{l}, t_b, t_e, X, (y = a) \rangle \mid \theta$ and in addition, the action a in $\mathcal{T}$ produces v.

4 LS^2: Syntax and Semantics

In this section, we describe the syntax and semantics of the logic LS^2.

4.1 Syntax of LS^2

The formulas of LS^2, together with their intuitive meanings, are shown in Figure 4. Action predicates, denoted R, capture reductions of the timed trace. There is one action predicate for each possible action. The second argument of each action predicate specifies a time point for the action. The predicate holds if a reduction corresponding to the action happened on the trace at the specified time point.

Other predicates (denoted M) capture other properties of the trace. $\mathsf{MemContents}(t, l, v)$ means that location l contains value v at time t. $\mathsf{Reset}(m, t, X)$ means that machine m was reset at time t, creating a new thread with id X (using the reduction rule (reset)). The two predicates of different arity named Honest capture honesty invariants described earlier. The predicate $t \geq t'$ represents algebraic ordering between time points. $e = e'$ represents syntactic equality between expressions. We also allow equality between time points. Together, the two time comparison operators $\geq$ and $=$ are enough to define all the other comparison operators: $\leq$, $<$, $>$ and $\neq$ between time points. In the sequel, we use these operators as if they were part of the formal syntax.

Formulas, denoted A, B, etc, are either predicates or constructed using connectives of first-order classical logic. The quantifiers $\forall x.A$ and $\exists x.A$ may range over expressions, thread ids, locations, and time points, but not over processes or machines.

The subjects of deduction in the logic are judgments, J. LS^2 has two new judgments that are intuitively explained below:

- $\vec{l}, t_b, t_e \vdash [P]_X\ A$: If, starting in a configuration that locks all locations in $\vec{l}$ for X, all reductions of X between t_b and t_e match P, then A holds.

- $\vec{l}, t_b, t_e \vdash^y [a]_X\ A$: If, starting in a configuration that locks all locations in $\vec{l}$ for X, the process of X executes exactly one action matching $y = a$ between t_b and t_e, then A holds.

As an invariant, logical deduction keeps t_b and t_e parametric. The only free variables allowed in A are t_b, t_e, and any variables free in P. In the second judgment, A may also mention y. This departs from PCL, in which A may also mention variables bound in P. This choice is purely a matter of style, one could reconstruct LS^2 allowing bound variables of P to occur in A.

4.2 Semantics of LS^2

We define the formal semantics of LS^2 by defining satisfaction of formulas and judgments over timed traces. We write $\mathcal{T} \models J$ to mean that $\mathcal{T}$ satisfies the judgment J. If $J = (\vdash A)$, we often abbreviate $\mathcal{T} \models J$ to $\mathcal{T} \models A$.

$$
\begin{array}{llll}
\text{Action Predicates} & R & ::= & \mathsf{Extend}(X,t,PCR(x),v) \mid \mathsf{Read}(X,t,l,v) \mid \mathsf{Write}(X,t,l,v) \mid \mathsf{Call}(X,t,c) \\
 & & \mid & \mathsf{Receive}(X,t,v) \mid \mathsf{Sign}(X,t,v) \mid \mathsf{Verify}(X,t,v,K) \mid \mathsf{Lock}(X,t,l) \mid \mathsf{Unlock}(X,t,l) \\
\text{Other Predicates} & M & ::= & \mathsf{MemContents}(t,l,v) \mid \mathsf{Reset}(m,t,X) \mid \mathsf{Honest}(\hat{X}) \mid \mathsf{Honest}(\hat{X},\vec{P}) \mid \ t \geq t' \ \mid \ e = e' \\
\text{Formulas} & A,B & ::= & R \mid M \mid A \wedge B \mid A \vee B \mid A \supset B \mid \neg A \mid \forall x.A \mid \exists x.A \\
\text{Judgments} & J & ::= & \vdash A \ \mid \ \vec{l},t_b,t_e \vdash [P]_X \ A \ \mid \ \vec{l},t_b,t_e \vdash^y [a]_X \ A
\end{array}
$$

Figure 4. Syntax of LS^2

Action Predicates

An action predicate is satisfied by $\mathcal{T}$ if $\mathcal{T}$ contains a reduction matching the corresponding action.

$\mathcal{T} \models \mathsf{Extend}(X,t,PCR(x),v,v')$ if in $\mathcal{T}$, X executed the action $\_ = \texttt{extend}\ PCR(x), l$ at time t, l contained v and $PCR(x)$ contained v' before the action.

$\mathcal{T} \models \mathsf{Read}(X,t,l,v)$ if in $\mathcal{T}$, X executed the action $x = \texttt{read}\ l$ at time t, receiving v from location l into x.

$\mathcal{T} \models \mathsf{Write}(X,t,l,v,v')$ if in $\mathcal{T}$, X executed the action $\_ = \texttt{write}\ l,v$ at time t and l contained v' before the action.

$\mathcal{T} \models \mathsf{Call}(X,t,c)$ if in $\mathcal{T}$, X executed the action $\_ = \texttt{call}\ c$ at time t.

$\mathcal{T} \models \mathsf{Send}(X,t,v)$ if in $\mathcal{T}$, X executed the action $\_ = \texttt{send}\ v$ at time t.

$\mathcal{T} \models \mathsf{Receive}(X,t,v)$ if in $\mathcal{T}$, X executed the action $x = \texttt{receive}$ at time t, receiving v into x.

$\mathcal{T} \models \mathsf{Sign}(X,t,v)$ if in $\mathcal{T}$, X executed the action $x = \texttt{sign}\ v$ at time t, receiving $SIG_{\hat{X}}\{|v|\}$ into x.

$\mathcal{T} \models \mathsf{Verify}(X,t,v,K)$ if in $\mathcal{T}$, X executed the action $\_ = \texttt{verify}\ s,v,K$ at time t, and $s = SIG_{\hat{K}}\{|v|\}$.

$\mathcal{T} \models \mathsf{Lock}(X,t,l)$ if in $\mathcal{T}$, X executed the action $\_ = \texttt{lock}\ l$ at time t.

$\mathcal{T} \models \mathsf{Unlock}(X,t,l)$ if in $\mathcal{T}$, X executed the action $\_ = \texttt{unlock}\ l$ at time t.

Other Predicates

Let $\mathcal{T} = C_0 \xrightarrow{t_1} C_1 \ldots \xrightarrow{t_n} C_n$. For uniformity, define $t_0 = -\infty$. Then,

$\mathcal{T} \models \mathsf{MemContents}(t,l,v)$ if $t_i < t \leq t_{i+1}$ for some $0 \leq i < n$ and in C_i, $\sigma(l) = v$, or $t > t_n$ and in C_n, $\sigma(l) = v$.

$\mathcal{T} \models \mathsf{Reset}(m,t,X)$ if in $\mathcal{T}$, the reduction (reset) happened on machine m at time t, creating a new thread with id X and process $Booting(m)$.

$\mathcal{T} \models \mathsf{Honest}(\hat{X})$ if $\hat{X}$ is assumed to be honest.

$\mathcal{T} \models \mathsf{Honest}(\hat{X},\vec{P})$ if $\hat{X}$ is honest, and in C_0, each process of $\hat{X}$ is in $\vec{P}$.

$\mathcal{T} \models t \geq t'$ if $t \geq t'$ (algebraically)

$\mathcal{T} \models e = e'$ if e and e' are syntactically equal.

Formulas

Satisfaction for formulas built with the connectives of first-order logic is defined in the obvious way. For example,

$\mathcal{T} \models A \wedge B$ if $\mathcal{T} \models A$ and $\mathcal{T} \models B$.

Judgments

Satisfaction for judgments captures their intuitive meaning described earlier.

$\mathcal{T} \models (\vdash A)$ if $\mathcal{T} \models A$

$\mathcal{T} \models (\vec{l},t_b,t_e \vdash [P]_X \ A)$ if for any ground time points t'_b and t'_e, $\mathcal{T} \gg \langle \vec{l},t'_b,t'_e,X,P \rangle \mid \theta$ implies $\mathcal{T} \models (A\theta)[t'_b/t_b][t'_e/t_e]$

$\mathcal{T} \models (\vec{l},t_b,t_e \vdash^y [a]_X \ A)$ if for any ground time points t'_b and t'_e, $\mathcal{T} \gg \langle \vec{l},t'_b,t'_e,X,y,a \rangle \mid \theta, v/y$ implies $\mathcal{T} \models (A\theta)[v/y][t'_b/t_b][t'_e/t_e]$

4.3 SRTM: Correctness Property

We formalize a correctness property for the modified SRTM protocol described in Figure 1. Later, in Section 5.2, we use the logic's proof system to actually establish this property. Suppose that the verifier's thread, V, executes the code $Verifier(m)$ in the time interval $[t_b,t_e)$. Then, the following four properties are guaranteed: (1) At some point of time t_R ($t_R < t_e$), the machine m was reset, (2) At some time point t_{Re} ($t_R < t_{Re} < t_e$), the TPM of m read $seq(0,BL(m),OS(m))$ from $m.PCR(s)$, (3) At some point of time t_{BL} ($t_R < t_{BL} < t_{Re}$), the correct

boot loader $BL(m)$ was called on m, and (4) Machine m was not reset between t_R and t_{Re}.

Formally, let us define the following four formulas corresponding to these four properties.

1. $A_1(t_R, X) = \mathsf{Reset}(m, t_R, X)$
2. $A_2(t_{Re}) = \exists TPM(m).\ \mathsf{Read}(TPM(m), t_{Re}, m.PCR(s), seq(0, BL(m), OS(m))$
3. $A_3(t_{BL}, X) = \mathsf{Call}(X, t_{BL}, BL(m))$
4. $A_4(t_R, t_{Re}) = \forall t.\forall Y.\ (t_R < t < t_{Re}) \supset \neg\mathsf{Reset}(t, m, Y)$

Then, we establish the following judgment (called J_{SRTM} in the sequel):

$$\cdot, t_b, t_e \vdash [Verifier(m)]_V\ \exists t_R, t_{Re}, t_{BL}, X.\ ((t_R < t_{BL} < t_{Re} < t_e) \wedge A_1(t_R, X) \wedge A_2(t_{Re}) \wedge A_3(t_{BL}, X) \wedge A_4(t_R, t_{Re}))$$

Several desirable properties of the protocol do not actually hold. These are discussed in Section 5.2.

5 LS^2: Proof System

The proof system is presented in Figures 5 and 6 respectively. In addition to these rules and axioms, we assume a full axiomatization of first-order classical logic with explicit equality on terms. We omit these axioms, since any presentation suffices. We also assume (implicitly) that the order on time points is total, i.e, it is reflexive, transitive, anti-symmetric, and that for any two time points t_1 and t_2, $(t_1 \geq t_2) \vee (t_2 \geq t_1)$ (totality). For the purpose of reasoning in the proof system, we do not need to assume that time points form a dense set, but we need this property for proving soundness (hence the choice of real numbers for representing time).

We use some notation to simplify our presentation. If $P = (x_1 = a_1; \ldots x_n = a_n)$ is a process, we define the *initial sequences* of P, written $IS(P)$, as the set of sequences of the form $x_1 = a_1; \ldots x_i = a_i$, where $0 \leq i \leq n$. This also includes the empty sequence. This easily generalizes to sets of processes; $IS(\vec{P})$ represents the set of prefixes of all processes in $\vec{P}$.

If a is an action, $R(X, t, x, a)$ denotes its corresponding action predicate, with X as the first argument, t as the second, and the remaining arguments taken from $x = a$. For example, if $a = \texttt{write}\ l, v$, then $R(X, t, x, a) = \mathsf{Write}(X, t, l, v)$. Similarly, if $a = \texttt{read}\ l$, then $R(X, t, x, a) = \mathsf{Read}(X, t, l, x)$. We write $\mathsf{NoReset}(m, t, t')$ to mean that m was not reset between t and t' (t included), i.e., as an abbreviation for $\forall t''.\forall Y.\ (t \leq t'' < t') \supset \neg\mathsf{Reset}(\mathsf{m}, \mathsf{t}'', \mathsf{Y})$. A slightly weaker property written $\mathsf{NoReset}^w(m, t, t')$ does not include the time point t. $\mathsf{NoReset}^w(m, t, t') = \forall t''.\forall Y.\ (t < t'' < t') \supset \neg\mathsf{Reset}(\mathsf{m}, \mathsf{t}'', \mathsf{Y})$.

We briefly describe the important rules and axioms. Rule (Seq) states that we can reason about a process $x = a; P$, all of whose actions happened between t_b and t_e, by assuming a time point t_m, such that action a happened before t_m and all other actions in P happened afterward, and combining the facts deduced from the two components. Rules (Lock) and (Unlock) are similar, except that the locations locked for the thread change as we move from the conclusion to the premises.

Rules (Honesty) and (Reset) allow us to reason with invariants: if it is known that a thread X can only be executing a fixed set of processes ($\{Booting(m)\}$ in case of (Reset)), and on every prefix of these processes, A holds, then A must hold. Axiom (Act) states that if thread X executes action a between t_b and t_e, then there is exactly one time point t in that interval on which a occurred.

Axiom (Mem1) states that if the only action of X in the interval $[t_b, t_e)$ does not write, extend or unlock l, at t_b, l contained v, X has a write lock on l, and the machine does not reset in $[t_b, t_e)$, then throughout the interval l has value v. Axioms (Mem2) and (Mem3) generalize (Mem1) when the action of X writes or extends l. In these cases, there is a time point t up to which l contains its old value v, and after which l contains the new value written by the action. Axioms (Mem4) – (Mem6) are similar to axioms (Mem1) – (Mem3), but they are used to reason about a thread which is started after a reset.

Axiom (VER) states that if a thread of agent $\hat{X}$ successfully verifies another honest agent $\hat{K}$'s signature, then at some earlier time some thread of $\hat{K}$ either sent the signature in a message, or stored it in memory. Axiom (PCR1) says that if a PCR contains $seq(0, v_1, \ldots, v_n)$ at some point of time, then at some earlier time, it must also have contained $seq(0, v_1, \ldots, v_{n-1})$, and that there was no reset in between (which would have set the PCR to 0). Axiom (PCR2) is similar.

5.1 Soundness

The main technical result of our work is a *soundness* theorem: every judgment provable in the proof system of LS^2 is satisfied by every timed trace. We prove a more general result. Let Γ denote a set of formulas, and let $\Gamma \Longrightarrow J$ mean that J can be proved assuming that each formula in Γ is provable. Let $\Gamma \models J$ mean that every trace which satisfies each formula in Γ also satisfies J.

$$\frac{\vec{l},t_b,t_m \vdash^y [a]_X\ A_1 \qquad \vec{l},t_m,t_e \vdash [P]_X\ A_2 \qquad a \neq \texttt{lock}\ l', \texttt{unlock}\ l' \qquad (t_m \text{ fresh})}{\vec{l},t_b,t_e \vdash [y = a; P]_X\ \exists t_m.\exists y.((t_b < t_m < t_e) \wedge A_1 \wedge A_2)}(\text{Seq})$$

$$\frac{\vec{l},t_b,t_m \vdash^y [\texttt{lock}\ l']_X\ A_1 \qquad \vec{l}+\{l'\},t_m,t_e \vdash [P]_X\ A_2 \qquad (t_m \text{ fresh})}{\vec{l},t_b,t_e \vdash [y = \texttt{lock}\ l'; P]_X\ \exists t_m.\exists y.((t_b < t_m < t_e) \wedge A_1 \wedge A_2)}(\text{Lock})$$

$$\frac{\vec{l}+\{l'\},t_b,t_m \vdash^y [\texttt{unlock}\ l']_X\ A_1 \qquad \vec{l},t_m,t_e \vdash [P]_X\ A_2 \qquad (t_m \text{ fresh})}{\vec{l}+\{l'\},t_b,t_e \vdash [y = \texttt{unlock}\ l'; P]_X\ \exists t_m.\exists y.((t_b < t_m < t_e) \wedge A_1 \wedge A_2)}(\text{Unlock})$$

$$\frac{\vdash A}{\vec{l},t_b,t_e \vdash [P]_X\ A}(\text{Nec1}) \qquad \frac{\vdash A}{\vec{l},t_b,t_e \vdash^y [a]_X\ A}(\text{Nec2})$$

$$\frac{\vec{l},t_b,t_e \vdash [P]_X\ A_1 \qquad \vec{l},t_b,t_e \vdash [P]_X\ A_2}{\vec{l},t_b,t_e \vdash [P]_X\ A_1 \wedge A_2}(\text{Conj1}) \qquad \frac{\vec{l},t_b,t_e \vdash^y [a]_X\ A_1 \qquad \vec{l},t_b,t_e \vdash^y [a]_X\ A_2}{\vec{l},t_b,t_e \vdash^y [a]_X\ A_1 \wedge A_2}(\text{Conj2})$$

$$\frac{\vec{l},t_b,t_e \vdash [P]_X\ A_1 \supset A_2 \qquad \vec{l},t_b,t_e \vdash [P]_X\ A_1}{\vec{l},t_b,t_e \vdash [P]_X\ A_2}(\text{Imp1}) \qquad \frac{\vec{l},t_b,t_e \vdash^y [a]_X\ A_1 \supset A_2 \qquad \vec{l},t_b,t_e \vdash^y [a]_X\ A_1}{\vec{l},t_b,t_e \vdash^y [a]_X\ A_2}(\text{Imp2})$$

$$\frac{\forall \rho \in IS(\vec{P}).\ (mustlocked(\rho),t_b,t_e \vdash [\rho]_X\ A)}{\vdash \mathsf{Honest}(\hat{X},\vec{P}) \supset \forall t_e.\ A[-\infty/t_b]}(\text{Honesty}) \qquad \frac{\forall \rho \in IS(Booting(m)).\ (locs(m),t_b,t_e \vdash [\rho]_X\ A)}{\vdash \mathsf{Reset}(m,t,X) \supset \forall t_e.\ (t_e > t) \supset A[t/t_b]}(\text{Reset})$$

Figure 5. Proof system (Rules) for LS^2

The following theorem states that $\Gamma \Longrightarrow J$ implies $\Gamma \models J$. Proof of the theorem proceeds by induction over the derivation of the given judgment, J (see Appendix A).

Theorem 1 (Soundness). *$\Gamma \Longrightarrow J$ implies $\Gamma \models J$.*

5.2 SRTM: Proof of Correctness

We now illustrate the proof system of the logic by proving the correctness property J_{SRTM} (Section 4.3) for the SRTM protocol. In order to prove the property, we have to make some assumptions. The assumptions are stated as a set Γ_{SRTM}:

$$\Gamma_{SRTM} \ = \ (\hat{V} \neq TP\hat{M}(m)),\ \mathsf{Honest}(TP\hat{M}(m), Signer(m))$$

The first assumption says that the verifier is not the same as the TPM of m, while the second assumption states that the TPM does not leak its signing key, and executes only the process $Signer(m)$. We prove the following theorem:

Theorem 2 (Correctness of SRTM). *If Γ_{SRTM} denotes the set defined above, and J_{SRTM} is the judgment defined in Section 4.3, then $\Gamma_{SRTM} \Longrightarrow J_{SRTM}$.*

A complete proof of the above theorem is described in Appendix B. We describe here, in brief, the major steps in the proof. Let $v_0 = seq(0, BL(m), OS(m))$. Suppose that thread V completely executes the process $Verifier(m)$ in the interval $[t_b, t_e)$. Then, since this process contains a verification step (last line), there must be a time point t_V ($t_V < t_e$) such that $\mathsf{Verify}(V, t_V, v_0, TP\hat{M}(m))$. By axiom (VER), there is some thread $TPM(m)$ of agent $TP\hat{M}(m)$ such that either $TPM(m)$ wrote $SIG_{TP\hat{M}(m)}\{|v_0|\}$ to some location in memory, or sent it in a message.

Next, we use the (Honesty) rule on thread $TPM(m)$, since we know from Γ_{SRTM} that it can only be executing the process $Signer(m)$. We show that: (1) $TPM(m)$ never writes any location, and (2) Whenever it sends the message $SIG_{TP\hat{M}(m)}\{|w|\}$, it reads w from $m.PCR(s)$. It follows from this, and our earlier deduction that at some time point t_{Re} ($t_{Re} < t_V$), $m.PCR(s)$ contained v_0. Using axioms (PCR1) and (PCR2), we deduce that there is a time point t_R ($t_R < t_{Re}$) such that the machine was reset at time t_R and not reset between t_R and t_{Re}.

Finally, we use the (Reset) rule to incorporate another invariant: for the process $Booting(m)$ started immediately after a reset, $m.PCR(s)$ can contain $seq(0, b, c, \ldots)$ at a later point only if in the interim, b was called. This uses axioms (Mem1) – (Mem6) and critically relies on the write locks on $m.PCR(s)$ and $m.bl\_loc$. It follows from this fact that at some time t_{BL} ($t_R < t_{BL} < t_{Re}$),

(Act) $\vec{l},t_b,t_e \vdash^y [a]_X\ \exists t.\ (t_b \le t < t_e) \wedge R(X,t,y,a) \wedge (\forall t'.\ ((t \ne t') \wedge (t_b \le t' < t_e)) \supset \neg R(X,t',y,a))$

(¬Act) $\vec{l},t_b,t_e \vdash^y [a]_X\ \forall t.\ (t_b \le t < t_e) \supset \neg R(X,t,x,a')$ $(a \ne a' \text{ or } x \ne y)$

(¬Act') $\vec{l},t_b,t_e \vdash [\,]_X\ \forall t.\ (t_b \le t < t_e) \supset \neg R(X,t,x,a)$

(Verify) $\vec{l},t_b,t_e \vdash^y [\texttt{verify}\ s,v,\hat{K}]_X\ s = SIG_{\hat{K}}\{|v|\}$

(Sign) $\vec{l},t_b,t_e \vdash^y [\texttt{sign}\ v]_X\ y = SIG_{\hat{X}}\{|v|\}$

(Mem=) $\vdash (\mathsf{MemContents}(t,l,v) \wedge \mathsf{MemContents}(t,l,v')) \supset (v = v')$

(Mem1)** $(l,\vec{l'}),t_b,t_e \vdash^y [a]_X\ (\mathsf{MemContents}(t_b,l,v) \wedge \mathsf{NoReset}(m,t_b,t_e)) \supset \forall t.\ (t_b < t \le t_e \supset \mathsf{MemContents}(t,l,v))$

(Mem1') $(l,\vec{l'}),t_b,t_e \vdash [\,]_X\ (\mathsf{MemContents}(t_b,l,v) \wedge \mathsf{NoReset}(m,t_b,t_e)) \supset \forall t.\ (t_b < t \le t_e \supset \mathsf{MemContents}(t,l,v))$

(Mem2) $(l,\vec{l'}),t_b,t_e \vdash^y [\texttt{write}\ l,v']_X\ (\mathsf{MemContents}(t_b,l,v) \wedge \mathsf{NoReset}(m,t_b,t_e)) \supset \exists t.\ ((t_b \le t < t_e) \wedge (\forall t'.\ (t < t' \le t_e) \supset \mathsf{MemContents}(t',l,v')) \wedge (\forall t'.\ (t_b < t' \le t) \supset \mathsf{MemContents}(t',l,v)))$

(Mem3) $(l,\vec{l'}),t_b,t_e \vdash^y [\texttt{extend}\ l,v']_X\ (\mathsf{MemContents}(t_b,l,seq(\vec{v})) \wedge \mathsf{NoReset}(m,t_b,t_e)) \supset \exists t.\ ((t_b \le t < t_e) \wedge (\forall t'.\ (t < t' \le t_e) \supset \mathsf{MemContents}(t',l,seq(\vec{v},v'))) \wedge (\forall t'.\ (t_b < t' \le t) \supset \mathsf{MemContents}(t',l,seq(\vec{v}))))$

(Mem4)** $(l,\vec{l'}),t_b,t_e \vdash^y [a]_X\ (\mathsf{Reset}(m,t_b,X) \wedge \mathsf{NoReset}^w(m,t_b,t_e)) \supset \forall t.\ (t_b < t \le t_e \supset \mathsf{MemContents}(t,l,0))$

(Mem4') $(l,\vec{l'}),t_b,t_e \vdash [\,]_X\ (\mathsf{Reset}(m,t_b,X) \wedge \mathsf{NoReset}^w(m,t_b,t_e)) \supset \forall t.\ (t_b < t \le t_e \supset \mathsf{MemContents}(t,l,0))$

(Mem5) $(l,\vec{l'}),t_b,t_e \vdash^y [\texttt{write}\ l,v']_X\ (\mathsf{Reset}(m,t_b,X) \wedge \mathsf{NoReset}^w(m,t_b,t_e)) \supset \exists t.\ ((t_b \le t < t_e) \wedge (\forall t'.\ (t < t' \le t_e) \supset \mathsf{MemContents}(t',l,v')) \wedge (\forall t'.\ (t_b < t' \le t) \supset \mathsf{MemContents}(t',l,0)))$

(Mem6) $(l,\vec{l'}),t_b,t_e \vdash^y [\texttt{extend}\ l,v']_X\ (\mathsf{Reset}(m,t_b,X) \wedge \mathsf{NoReset}^w(m,t_b,t_e)) \supset \exists t.\ ((t_b \le t < t_e) \wedge (\forall t'.\ (t < t' \le t_e) \supset \mathsf{MemContents}(t',l,seq(0,v'))) \wedge (\forall t'.\ (t_b < t' \le t) \supset \mathsf{MemContents}(t',l,0)))$

(VER) $\vdash (\mathsf{Verify}(X,t,v,\hat{K}) \wedge (\hat{X} \ne \hat{K}) \wedge \mathsf{Honest}(\hat{X})) \supset \exists K.\exists t'.\ (t' < t) \wedge (\mathsf{Send}(K,t',SIG_{\hat{K}}\{|v|\}) \vee \exists l.\ \mathsf{Write}(K,t',l,SIG_{\hat{K}}\{|v|\}))$

(READ) $\vdash \mathsf{Read}(X,t,l,v) \supset \mathsf{MemContents}(t,l,v)$

(PCR1) $\vdash \mathsf{MemContents}(t,m.PCR(s),seq(0,v_1,\ldots,v_n)) \supset \exists t'.\ (t' < t) \wedge \mathsf{MemContents}(t',m,seq(0,v_1,\ldots,v_{n-1})) \wedge \mathsf{NoReset}(t',t)$

(PCR2) $\vdash \mathsf{MemContents}(t,m.PCR(s),0) \supset \exists t'.\exists X.\ (t' < t) \wedge \mathsf{Reset}(m,t',X) \wedge \mathsf{NoReset}^w(m,t',t)$

(Hon) $\vdash \mathsf{Honest}(\hat{X},\vec{P}) \supset \mathsf{Honest}(\hat{X})$

** Side condition: $a \ne \texttt{write}\ l,v'$ and $a \ne \texttt{extend}\ l,v'$ and $a \ne \texttt{unlock}\ l$

Figure 6. Proof system (Axioms) for LS^2

$BL(m)$ was called. This completes the proof.

It is known that the SRTM protocol is vulnerable to Time-Of-Check-To-Time-Of-Use (TOCTTOU) attacks where code is modified after being measured, but before it is loaded [30]. We identify four such attacks based on our analysis. First, write locks are crucial for correctness. In the proof, this shows up in the last part, where we use the axioms (Mem1) – (Mem6) to infer invariants about the role $Booting(m)$. In actual practice, memory is not locked during booting, making the protocol susceptible to attacks.

Second, the protocol does not guarantee that the operating system was loaded, merely that it was measured. Formally, this shows up in the last part of the proof, where it is impossible to prove a stronger invariant for $Booting(m)$, since between the measurement and loading of the operating system (Figure 1, lines 4 and 6), another thread on the machine may signal the TPM to produce a signature and send it to the verifier. This problem extends to the actual protocol, where, there can be no guarantee that the last piece of code measured was actually loaded. One way to fix this problem is to use read-write locks, thus preventing the TPM from signing the PCR until the lock is released.

Third, the machine can be reset after the TPM signs the PCR. This shows up in the first part of the proof, where, by verifying the signature, all that the verifier can deduce is that at some point of time in the past, the TPM generated the signature. There is no information about recency. This makes our version of the protocol susceptible to man-in-the-middle replay attacks. The actual SRTM protocol includes a nonce intended to prevent this gap. However, even with the nonce, one can only prove that the machine was running the measured software *at the time of generation of the nonce*.

Finally, the proof of the protocol relies on the fact that PCRs can be reset only during the boot process (axiom (PCR2)). However, current hardware does not enforce this, and this makes the protocol insecure [20].

6. Related Work

As mentioned before, LS^2 shares a number of features with PCL [9] and therefore with other logics of programs [2, 15, 18]. 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 [22] 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.

In previous work, Abadi and Wobber used an authorization logic to describe the basic ideas of NGSCB, the predecessor to the TCG [1]. Their formalization documents and clarifies basic NGSCB concepts rather than proving specific properties of systems utilizing a TPM. Gurgens et al. used a model checker to analyze the security of several TCG protocols [14]. Millen et al. employed a model checker to understand the role and trust relationships of a system performing a remote attestation protocol [24]. Our analysis with LS^2 is a complementary approach: It provides provable guarantees beyond those provided by model checkers, but with a less fine-grained model. Chen et al. developed a formal logic tailored to the analysis of a remote attestation protocol and suggested improvements [3]. LS^2 is designed to be a more general logic with TCG protocols providing one set of applications. Lin [23] used a theorem prover and model finder to analyze of the security of the TPM against invalid sequences of API calls. Proving the security of systems with complex APIs such as the TPM is an active area of research and a possible future application of LS^2 [17].

7. Conclusion and Future Work

As a first step towards developing a theoretical basis for secure systems, we introduce LS^2, a logic for reasoning about security properties of systems with shared memory that communicate over a network. Our technical contributions include a precise definition of a programming language for modeling such systems, a logic for specifying properties, and a sound proof system for reasoning about such systems. We apply the logic to establish a non-trivial property of a part of the TCG remote attestation protocol. Our analysis clearly identifies the ways in which the protocol may fail, thereby providing a formal justification for previously discovered attacks [20, 30].

In future work, we plan to use this logic to carry out a detailed analysis of the TCG's SRTM and sealed storage protocols and extensions of these protocols [19]. In a complementary effort, we are currently investigating semantically well-founded methods for modeling and analysis of secure system implementations. Our long term goal is to develop a coherent framework for analysis of secure system architectures and implementations by meaningfully combining the two efforts.

References

[1] M. Abadi and T. Wobber. A logical account of ngscb. In *Proceedings of Formal Techniques for Networked and Distributed Systems*, 2004.

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

[3] 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.

[4] A. Datta, A. Derek, J. C. Mitchell, and D. Pavlovic. A derivation system for security protocols and its logical formalization. In *Proceedings of 16th IEEE Computer Security Foundations Workshop*, pages 109–125. IEEE, 2003.

[5] A. Datta, A. Derek, J. C. Mitchell, and D. Pavlovic. Secure protocol composition (extended abstract). In *Proceedings of ACM Workshop on Formal Methods in Security Engineering*, pages 11–23, 2003.

[6] A. Datta, A. Derek, J. C. Mitchell, and D. Pavlovic. Abstraction and refinement in protocol derivation. In *Proceedings of 17th IEEE Computer Security Foundations Workshop*, pages 30–45. IEEE, 2004.

[7] A. Datta, A. Derek, J. C. Mitchell, and D. Pavlovic. Secure protocol composition. In *Proceedings of 19th Annual Conference on Mathematical Foundations of Programming Semantics*. ENTCS, 2004.

[8] A. Datta, A. Derek, J. C. Mitchell, and D. Pavlovic. A derivation system and compositional logic for security protocols. *Journal of Computer Security*, 13(3):423–482, 2005.

[9] A. Datta, A. Derek, J. C. Mitchell, and A. Roy. Protocol composition logic (pcl). *Electr. Notes Theor. Comput. Sci.*, 172:311–358, 2007.

[10] A. Datta, A. Derek, J. C. Mitchell, V. Shmatikov, and M. Turuani. Probabilistic polynomial-time semantics for a protocol security logic. In *Proceedings of the 32nd International Colloquium on Automata, Languages and Programming (ICALP '05)*, Lecture Notes in Computer Science, pages 16–29. Springer-Verlag, 2005.

[11] A. Datta, A. Derek, J. C. Mitchell, and B. Warinschi. Computationally sound compositional logic for key exchange protocols. In *Proceedings of 19th IEEE Computer Security Foundations Workshop*, pages 321–334. IEEE, 2006.

[12] N. Durgin, J. C. Mitchell, and D. Pavlovic. A compositional logic for protocol correctness. In *Proceedings of 14th IEEE Computer Security Foundations Workshop*, pages 241–255. IEEE, 2001.

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

[14] 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.

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

[16] C. He, M. Sundararajan, A. Datta, A. Derek, and J. C. Mitchell. A modular correctness proof of IEEE 802.11i and TLS. In *CCS '05: Proceedings of the 12th ACM Conference on Computer and Communications Security*, pages 2–15, 2005.

[17] J. Herzog. Applying protocol analysis to security device interfaces. *IEEE Security and Privacy*, 4(4):84–87, Jul/Aug 2006.

[18] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

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

[20] B. Kauer. OSLO: Improving the security of trusted computing. In *Proceedings of the USENIX Security Symposium*, Aug. 2007.

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

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

[23] A. H. Lin. Automated analysis of security apis. Master's thesis, Massachusetts Institute of Technology, 2005.

[24] J. Millen, J. Guttman, J. Ramsdell, J. Sheehy, and B. Sniffen. Analysis of a measured launch. Technical report, The MITRE Corporation, 2007.

[25] A. Pnueli. The temporal logic of programs. In *Proceedings of 19th Annual Symposium on Foundations on Computer Science*, 1977.

[26] A. Roy, A. Datta, A. Derek, J. C. Mitchell, and J.-P. Seifert. Secrecy analysis in protocol composition logic., 2006. to appear in Proceedings of 11th Annual Asian Computing Science Conference, December 2006.

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

[28] 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.

[29] 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.

[30] E. Shi, A. Perrig, and L. van Doorn. BIND: A time-of-use attestation service for secure distributed systems. In *Proceedings of IEEE Symposium on Security and Privacy*, May 2005.

[31] Trusted Computing Group (TCG). `https://www.trustedcomputinggroup.org/`, 2003.

A Proof of Soundness

In this appendix we prove soundness. The reader may recall that we are trying to prove that $\Gamma \Longrightarrow J$ implies $\Gamma \models J$. We do this by induction on derivation of $\Gamma \Longrightarrow J$ in the proof system. We analyze cases of the last rule in the proof, and show some representative cases below.

Case (HYP). This is the case where $\Gamma \Longrightarrow A$ because $A \in \Gamma$. We need to show that $\mathcal{T} \models \Gamma$ implies $\mathcal{T} \models (\vdash A)$. This follows because $A \in \Gamma$.

Case (Seq).

$$\frac{\begin{array}{c}\vec{l}, t_b, t_m \vdash^y [a]_X\ A_1 \qquad \vec{l}, t_m, t_e \vdash [P]_X\ A_2 \\ a \neq \texttt{lock}\ l', \texttt{unlock}\ l' \qquad (t_m \text{ fresh})\end{array}}{\vec{l}, t_b, t_e \vdash [y = a; P]_X\ \exists t_m. \exists y. ((t_b < t_m < t_e) \wedge A_1 \wedge A_2)}$$

Suppose for some ground time point t'_b and t'_e, $\mathcal{T} \gg \langle \vec{l}, t'_b, t'_e, X, (y = a; P)\rangle \mid \theta$. By definition, in $\mathcal{T}$, there is a configuration C_i containing the thread $\langle X, (y = a; P; Q)\theta\rangle$, and in some later C'_i (reached at some time less than t'_e), there is a thread $\langle X, (Q\theta)(\varphi[y \mapsto v])\rangle$. Clearly, then at some time point t_a between C_i and C'_i, there is a reduction $C_a \xrightarrow{t_a} C'_a$ that reduces $y = a$ and substitutes v for y, and the remaining reductions of P produce φ. Let t_n be the time at which the next reduction before t_e occurs in X. (If there is no such reduction, P is empty; choose $t_n = t_e$.) Now take $t'_m = (t_a + t'_m)/2$. Clearly, then $\mathcal{T} \gg \langle \vec{l}, t'_b, t'_m, X, y, a\rangle \mid \theta, v/y$ and $\mathcal{T} \gg \langle \vec{l}, t'_m, t'_e, X, P\rangle \mid \theta, v/y$. Hence, by i.h., $\mathcal{T} \models (A_1\theta)[v/y][t'_b/t_b][t'_m/t_m]$, and $\mathcal{T} \models (A_2\theta)[v/y][t'_m/t_m][t'_e/t_e]$. Thus, $\mathcal{T} \models ((A_1 \wedge A_2)\theta)[v/y][t'_b/t_b][t'_m/t_m][t'_e/t_e]$. We immediately have $\mathcal{T} \models (\ \exists t_m. \exists y. ((t_b < t_m < t_e) \wedge A_1 \wedge A_2))\theta[t'_b/t_b][t'_e/t_e]$, as required, since $t'_b < t'_m < t'_e$.

Cases (Lock) and (Unlock). Similar to the above case.

Case (Nec1).

$$\frac{\vdash A}{\vec{l}, t_b, t_e \vdash [P]_X\ A}(\text{Nec1})$$

Suppose for some t'_b, t'_e, $\mathcal{T} \gg \langle \vec{l}, t'_b, t'_e, X, P\rangle \mid \theta$. We need to show that $\mathcal{T} \models (A\theta)[t'_b/t_b][t'_e/t_e]$. However, since $\vdash A$, A must be closed. Hence $(A\theta)[t'_b/t_b][t'_e/t_e] = A$. Thus it is enough to show that $\mathcal{T} \models A$. This follows immediately from the induction hypothesis.

Case (Nec2). Similar to above case.

Cases (Conj1), (Conj2), (Imp1), (Imp2). These follow from the definition of satisfaction for $A_1 \wedge A_2$ and $A_1 \supset A_2$.

Case (Honesty).

$$\frac{\forall \rho \in IS(\vec{P}).\ (\mathit{mustlocked}(\rho), t_b, t_e \vdash [\rho]_X\ A)}{\vdash \mathsf{Honest}(\hat{X}, \vec{P}) \supset \forall t_e.\ A[-\infty/t_b]}(\text{Honesty})$$

We have to show $\mathcal{T} \models \mathsf{Honest}(\hat{X}, \vec{P}) \supset \forall t_e.\ A[-\infty/t_b]$. So, suppose that $\mathcal{T} \models \mathsf{Honest}(\hat{(X)}, \vec{P})$, and pick any ground time point t'_e. It suffices to show that $\mathcal{T} \models A[-\infty/t_b][t'_e/t_e]$. Now take *any* thread X in C_0 that belongs to $\hat{X}$. If there is no such thread, we can assume an empty process. Then the process of this thread is in $\vec{P}$. Let P be this process, and suppose that up to time t_e (but not including it), a prefix P' (possibly empty) has reduced, i.e., $P = P'; P''$ and P' reduces in the interval $(-\infty, t_e)$. It follows immediately that $\mathcal{T} \gg \langle \mathit{mustlocked}(P'), -\infty, t'_e, X, P'\rangle \mid \cdot$. (The fact that $\mathit{mustlocked}(P')$ is locked for X follows from well-formedness of configurations.) Clearly, $P' \in IS(\vec{P})$. Hence, by i.h., $\mathcal{T} \models A[-\infty/t_b][t'_e/t_e]$, as required.

Case (Reset).

$$\frac{\forall \rho \in IS(\mathit{Booting}(m)).\ (\mathit{locs}(m), t_b, t_e \vdash [\rho]_X\ A)}{\vdash \mathsf{Reset}(m, t, X) \supset \forall t_e.\ (t_e > t) \supset A[t/t_b]}(\text{Reset})$$

This is similar to the above case, except that instead of $-\infty$, we use t.

Case (Act).

$$\vec{l}, t_b, t_e \vdash^y [a]_X\ \exists t.\ (t_b \leq t < t_e) \wedge R(X, t, y, a) \wedge (\forall t'.\ ((t \neq t') \wedge (t_b \leq t' < t_e)) \supset \neg R(X, t', y, a))$$

Suppose that for some t'_b, t'_e, $\mathcal{T} \gg \langle \vec{l}, t'_b, t'_e, X, y, a\rangle \mid \theta, v/y$. We have to show that $\mathcal{T} \models \exists t.\ (t'_b \leq t < t'_e) \wedge R(X, t, y, a) \wedge (\forall t'.\ ((t \neq t') \wedge (t'_b \leq t' < t_e;)) \supset \neg R(X, t', y, a))\theta[v/y]$. By the definition of the match, there has to be a time point t ($t_b \leq t < t'_e$) such that $a\theta$ happened at t (hence $R(X, t, y, a)\theta[v/y]$ holds), and that no other action of X happened at any other time in the interval $[t'_b, t'_e)$. This is what we had to show.

Cases (¬Act) and (¬Act'). These are similar to the previous case.

Cases (Verify) and (Sign). These follow directly from the reduction rules.

Case (Mem=). We must show that $\mathcal{T} \models (\mathsf{MemContents}(t,l,v) \wedge \mathsf{MemContents}(t,l,v')) \supset (v = v')$. So suppose that $\mathcal{T} \models \mathsf{MemContents}(t,l,v)$ and $\mathcal{T} \models \mathsf{MemContents}(t,l,v')$. By definition, at time t, in $\mathcal{T}$, $\sigma(l) = v$ and $\sigma(l) = v'$. But σ is a function, so $v = v'$.

Case (Mem1).

$$(l,\vec{l'}),t_b,t_e \vdash^y [a]_X\ (\mathsf{MemContents}(t_b,l,v) \wedge \mathsf{NoReset}(m,t_b,t_e)) \supset \forall t.\ (t_b < t \leq t_e \supset \mathsf{MemContents}(t,l,v))$$

Suppose that for some t'_b, t'_e, $\mathcal{T} \gg \langle (l,\vec{l'}),t'_b,t'_e,X,y,a\rangle \mid \theta, v'/y$. Assume that $\mathcal{T} \models \mathsf{MemContents}(t_b,l,v)\theta[v'/y][t'_b/t_b][t'_e/t_e]$ and $\mathcal{T} \models \mathsf{NoReset}(m,t_b,t_e)\theta[v'/y][t'_b/t_b][t'_e/t_e]$. These imply that $\mathcal{T} \models \mathsf{MemContents}(t'_b,l,v)$ and $\mathcal{T} \models \mathsf{NoReset}(m,t'_b,t'_e)$. The first of these means that in $\mathcal{T}$, at time t'_b, $\sigma(l) = v$. Now, there are only three ways to change the value in a memory location: extend it, write to it, or reset the machine. However, we know from the side condition that the only action of X in $[t'_b,t'_e)$, namely a, neither writes, nor extends l. Also from the condition $\mathsf{NoReset}(t'_b,t'_e,m)$, the machine was not reset in the same interval. Furthermore, since at t'_b, l is locked for X, and a does not unlock it, it follows that no other thread could have changed the value in l. Thus the earliest point at which l could be changed is t'_e, and hence in the interval $(t_b,t_e]$, l must contain v, or equivalently, $\forall t.\ (t_b < t \leq t_e \supset \mathsf{MemContents}(t,l,v))$.

Cases (Mem1') – (Mem6). These are similar to the above case.

Case (VER).

$$\vdash (\mathsf{Verify}(X,t,v,\hat{K}) \wedge (\hat{X} \neq \hat{K}) \wedge \mathsf{Honest}(\hat{X})) \supset \exists K.\exists t'.\ (t' < t) \wedge (\mathsf{Send}(K,t',SIG_{\hat{K}}\{|v|\}) \vee \exists l.\ \mathsf{Write}(K,t',l,SIG_{\hat{K}}\{|v|\}))$$

Suppose $\mathcal{T} \models \mathsf{Verify}(X,t,v,\hat{K})$. It follows that at time t, X executes the action `verify` $SIG_{\hat{K}}\{|v|\},v,\hat{K}$ in $\mathcal{T}$. Since in the initial configuration, X cannot contain $SIG_{\hat{K}}\{|v|\}$ (because $\hat{K}$ is honest, and $\hat{X} \neq \hat{K}$, at some earlier time point $SIG_{\hat{K}}\{|v|\}$ must have appeared in X's thread for the first time. This could only have happened in two ways: either some other thread sent it to X, or X read it from a memory location. In the latter case, some other thread Y wrote it to the location. In either case, some other thread either sent the signature to X, or wrote it to memory at an earlier time. If this thread belongs to $\hat{K}$, we are done, else we can repeat the argument on thread Y.

Case (READ). Follows from definition of reduction.

Case (PCR1). Suppose $\mathcal{T} \models \mathsf{MemContents}(t,m.PCR(s),seq(0,v_1,\ldots,v_n))$. Then at time t, the contents of $m.PCR(s)$ were $seq(0,v_1,\ldots,v_n)$. Now a straightforward induction on the number of reductions in the trace shows that either $m.PCR(s)$ contained $seq(0,v_1,\ldots,v_n)$ in the initial configuration, or $0 = seq(0,v_1,\ldots,v_n)$, or at some earlier point of time $m.PCR(s)$ contained $seq(0,v_1,\ldots,v_{n-1})$, and there was no subsequent reset. The first two possibilities are ruled out by the definition of the initial configuration, and the fact that 0 is special and cannot equal a hash (a fundamental assumption made about PCRs). Thus the third possibility must be the case, as required.

Case (PCR2). Suppose $\mathcal{T} \models \mathsf{MemContents}(t,m.PCR(s),0)$. So at time t, $m.PCR(s)$ contained 0. Since in the initial configuration, $m.PCR(s)$ cannot contain 0, and no extension can place 0 in it, the only way this happened was by a reset. There may have been many resets; we choose the last one before t, and the time t' of this reset satisfies our required property.

Case (Hon). Follows from definition of honesty.

B Proof of Correctness of SRTM Protocol

A complete proof of the correctness of the SRTM protocol is shown in Figure 7. We remind the reader that the judgment we are trying to prove, J_{SRTM}, was defined in Section 4.3 as:

$$\cdot,t_b,t_e \vdash [Verifier(m)]_V\ \exists t_R,t_{Re},t_{BL},X.\ ((t_R < t_{BL} < t_{Re} < t_e) \wedge A_1(t_R,X) \wedge A_2(t_{Re}) \wedge A_3(t_{BL},X) \wedge A_4(t_R,t_{Re}))$$

where $A_1 - A_4$ are defined as follows:

1. $A_1(t_R,X) = \mathsf{Reset}(m,t_R,X)$
2. $A_2(t_{Re}) = \exists TPM(m).\ \mathsf{Read}(TPM(m),t_{Re},m.PCR(s), seq(0,BL(m),OS(m))$
3. $A_3(t_{BL},X) = \mathsf{Call}(X,t_{BL},BL(m))$

4. $A_4(t_R, t_{Re}) = \forall t.\forall Y.\ (t_R < t < t_{Re}) \supset \neg \mathsf{Reset}(t, m, Y)$

We assume that the following formulas are provable:

$\Gamma_{SRTM} \ = \ (\hat{V} \neq TP\hat{M}(m)),\ \mathsf{Honest}(TP\hat{M}(m), Signer(m))$

Figure 7 shows the 10 major steps used in the proof, together with the rules needed to conclude them. Each major step concludes a judgment of the form $\cdot, t_b, t_e \vdash [Verifier(m)]_V\ A$, where A is shown in the third column of the table.

	Axioms/Rules	Formula A in $\cdot, t_b, t_e \vdash [Verifier(m)]_V\, A \qquad (v_0 = seq(0, BL(m), OS(m)))$
1.	Act	$\exists t_V.\ (t_V < t_e) \wedge \mathsf{Verify}(V, t_V, v_0, \hat{TPM}(m))$
2.	VER	$\exists t'.\ (t' < t_e) \wedge (\exists TPM(m).\ Send(TPM(m), t', SIG_{\hat{TPM}(m)}\{\lvert v_0 \rvert\}) \vee \exists l.\ \mathsf{Write}(TPM(m), t', l, v_0))$
3.	Honesty, Act ¬Act, ¬Act'	$\forall t.\ (-\infty \leq t < t_e) \supset (\forall l. \forall v.\ \neg\mathsf{Write}(TPM(m), t, l, v))$ $\wedge\ (\forall w.\ \mathsf{Send}(TPM(m), t, SIG_{\hat{TPM}(m)}\{\lvert w \rvert\})$ $\supset \exists t_{Re}.\ ((t_{Re} < t) \wedge \mathsf{Read}(TPM(m), t_{Re}, m.PCR(s), w)))$
4.	(2), (3), Nec1, Imp1, Conj1	$\exists t_{Re}.\ (t_{Re} < t_e) \wedge ((\exists TPM(m).\ \mathsf{Read}(TPM(m), t_{Re}, m.PCR(s), v_0)) \equiv A_2(t_{Re}))$
5.	READ	$\exists t_{Re}.\ (t_{Re} < t_e) \wedge A_2(t_{Re}) \wedge\ \mathsf{MemContents}(t_{Re}, m.PCR(s), v_0)$
6.	PCR1	$\exists t_{Re}, t_2, t_1.\ (t_1 < t_2 < t_{Re} < t_e) \wedge A_2(t_{Re}) \wedge$ $\mathsf{MemContents}(t_{Re}, m.PCR(s), seq(0, BL(m), OS(m))) \wedge$ $\mathsf{MemContents}(t_2, m.PCR(s), seq(0, BL(m))) \wedge$ $\mathsf{MemContents}(t_1, m.PCR(s), seq(0)) \wedge$ $\mathsf{NoReset}(m, t_1, t_{Re})$
7.	PCR2	$\exists t_{Re}, t_2, t_1, t_R, X.\ (t_R < t_1 < t_2 < t_{Re} < t_e) \wedge A_2(t_{Re}) \wedge$ $\mathsf{MemContents}(t_{Re}, m.PCR(s), seq(0, BL(m), OS(m))) \wedge$ $\mathsf{MemContents}(t_2, m.PCR(s), seq(0, BL(m))) \wedge$ $\mathsf{MemContents}(t_1, m.PCR(s), seq(0)) \wedge$ $\mathsf{Reset}(m, t_R, X) \wedge$ $\mathsf{NoReset}^w(m, t_R, t_{Re})$
8.	Reset, Mem1 – Mem6, Act, ¬Act, Mem=	$\exists t_{Re}, t_R, X.\ (t_R < t_{Re} < t_e) \wedge A_2(t_{Re}) \wedge$ $\mathsf{MemContents}(t_{Re}, m.PCR(s), seq(0, BL(m), OS(m))) \wedge$ $\mathsf{Reset}(m, t_R, X) \wedge$ $\mathsf{NoReset}^w(m, t_R, t_{Re}) \wedge$ $\forall t'_e.\ (\mathsf{NoReset}^w(m, t_R, t'_e) \wedge \mathsf{Reset}(m, t_R, X))$ $\supset \forall t, b, \vec{o}.\ (t_R < t \leq t'_e)$ $\supset (\neg(\mathsf{MemContents}(t, m.PCR(s), seq(0, b, \vec{o})))$ $\vee\ (\exists t_{BL}.\ (t_R < t_{BL} < t) \wedge \mathsf{Call}(X, t_{BL}, b)))$
9.	Set $t'_e = t = t_{Re}$, $\vec{o} = OS(m)$, $b = BL(m)$	$\exists t_{Re}, t_R, X.\ (t_R < t_{Re} < t_e) \wedge A_2(t_{Re}) \wedge$ $\mathsf{MemContents}(t_{Re}, m.PCR(s), seq(0, BL(m), OS(m))) \wedge$ $\mathsf{Reset}(m, t_R, X) \wedge$ $\mathsf{NoReset}^w(m, t_R, t_{Re}) \wedge$ $(\mathsf{NoReset}^w(m, t_R, t_{Re}) \wedge \mathsf{Reset}(m, t_R, X))$ $\supset\ (t_R < t_{Re} \leq t_{Re})$ $\supset (\neg(\mathsf{MemContents}(t_{Re}, m.PCR(s), seq(0, BL(m), OS(m))))$ $\vee\ (\exists t_{BL}.\ (t_R < t_{BL} < t_{Re}) \wedge \mathsf{Call}(X, t_{BL}, BL(m))))$
10.	Simplify	$\exists t_R, t_{Re}, t_{BL}, X.\ (t_R < t_{BL} < t_{Re} < t_e) \wedge A_2(t_{Re}) \wedge$ $(\mathsf{Reset}(m, t_R, X) \equiv A_1(t_R, X)) \wedge$ $(\mathsf{NoReset}^w(m, t_R, t_{Re}) \equiv A_4(t_R, t_{Re})) \wedge$ $(\mathsf{Call}(X, t_{BL}, BL(m)) \equiv A_3(t_{BL}, X))$

Figure 7. Proof of correctness for the SRTM protocol