A Logic for Reasoning About Networked Secure Systems*

Deepak Garg Jason Franklin Dilsun Kaynar Anupam Datta

{dg,jfrankli,dilsun}@cs.cmu.edu, danupam@cmu.edu
Carnegie Mellon University

Abstract

We initiate a program to model and analyze end-to-end security properties of contemporary secure systems that rely on network protocols and memory protection. Specifically, this paper introduces the *Logic of Secure Systems* (LS^2). LS^2 extends an existing logic for security protocols by incorporating shared memory, time and limited forms of access control. The proof system for LS^2 supports high-level reasoning about secure systems in the presence of adversaries on the network and the local machine. We prove a soundness theorem for the proof system and illustrate its use by proving a relevant security property of a protocol inspired by the Transport Layer Protocol of the Secure Shell (SSH).

1 Introduction

We initiate a program to model and analyze end-to-end security properties of contemporary secure systems that rely on network protocols and memory protection. Examples of such systems include operating systems, hypervisors, virtual machine monitors and secure co-processor-based systems such as those utilizing the Trusted Computing Group's Trusted Platform Module (TPM) [1, 26, 27]. These systems are under attack from two classes of malicious agents: (a) *network adversaries* who intercept messages passed between protocol participants or inject fake messages, and (b) *local adversaries* in control of malicious local threads. A local adversary may perform a number of insidious attacks including altering local state on machines (both code and data), stealing secrets such as private keys, and exploiting TOCTTOU (time of check to time of use) race conditions [2, 3]. The two classes of adversaries may even collude; a malicious agent may, for example, insert a Trojan horse to steal keys, and listen on the network for messages at the same time. While the network adversary is successfully addressed by research on security protocol analysis, research into the design and analysis of secure systems with local adversaries is less mature. One common strategy to counter local adversaries is to enumerate and defend against specific attacks. However, the number and complexity of local attacks makes this process unsatisfactory. In this paper, we

*This work was partially supported by the Air Force Research Laboratory grant FA87500720028 Accountable Information Flow via Explicit Formal Proof, the U.S. Army Research Office contract on Perpetually Available and Secure Information Systems (DAAD19-02-1-0389) to CMU's CyLab, and the NSF Science and Technology Center TRUST. The second author performed this research while on appointment as a U.S. Department of Homeland Security (DHS) Fellow under the DHS Scholarship and Fellowship Program.

develop a unified security analysis method that takes into account the capabilities of both classes of adversaries.

Specifically, this paper introduces the *Logic of Secure Systems* (LS^2). The logic is built around a programming language for specifying the behavior of systems. It draws its lineage from a logic for network protocol analysis, *Protocol Composition Logic* (PCL) [6, 7, 10, 22]. We focus on the syntax, semantics, and general theory of LS^2 in this paper, leaving substantial applications to other papers.

Programming Language. 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*. It includes constructs to model cryptographic primitives, network communication and other common system motifs such as shared memory. The language mixes primitives from process calculi and both functional and imperative languages. It borrows from PCL standard process calculus style communication primitives as well as functional primitives for standard cryptographic operations. It extends PCL in three significant ways. First, the language includes a model of shared memory in the form of abstract locations and operations for reading and writing to these locations. This is essential for modeling many secure systems. The primitives for reading and writing locations are inspired by the treatment of memory cells in impure functional languages like Standard ML [17]. Second, we model *memory protection*, a fundamental building block for secure systems [23]. Protection is modeled at a high-level of abstraction – there are primitives to allow threads to acquire exclusive-write locks on memory locations that preclude other threads from writing to them, but we abstract over the actual implementation of these locks. Third, we explicitly model physical machines and assume that each memory location and each thread is located on a unique machine. This is motivated by an intent to reason about systems in the presence of machine failures and unexpected machine resets. The operational semantics of the programming language specifies how concurrent programs (representing the behavior of the system and the adversary) execute to produce traces.

Logic. The logic LS^2 is used to express and reason about properties of traces obtained from programs. LS^2 retains some basic features of PCL, such as predicates for reasoning about signatures and their verifications. However, LS^2 departs from PCL in the basic style of reasoning: instead of associating pre-conditions and post-conditions with all actions in a process (as PCL does), we model time explicitly (as a dense total order), and associate monotonically increasing time points with events on a trace. The basic modality, $[P]_I^{t_b,t_e}\ A$, means that in any trace where thread I executes exactly the actions in P during the interval $[t_b, t_e)$, formula A holds. The presence of explicit time allows us to express invariants about memory; for instance, we may express in LS^2 that the memory location l contains the same value v throughout the interval $[t_1, t_2]$. Explicit time is also used to reason about the relative order of events. Our formal treatment of explicit time is based on work by one of the authors and others [8], and also draws ideas from hybrid logics [4, 21]. 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 [20]. 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., [14, 24, 25]).

Security properties of a program are established without explicitly reasoning about adversarial actions using LS^2's proof system. A program independent *soundness theorem* guarantees that any property so established actually holds over all traces obtainable from the program, including those that contain any number of adversarial threads. This implicit treatment of adversaries considerably simplifies proofs. Designing a sound proof system that supports this local style of reasoning in spite of the global nature of memory changes turned out to be a significant challenge. Despite this difficulty, we believe that the final design of LS^2 is robust, modular, and amenable to application specific extensions. In particular, the proof of the soundness theorem is largely independent of both the cryptographic primitives and the operations on memory that are allowed in the programming language.

Application. As an illustrative example, we analyze a protocol inspired by SSH [28], a network protocol that allows data to be exchanged over a secure channel between a client and a server. SSH uses public-key cryptography to provide mutual authentication. Since SSH is often used with uncertified (or self-certified) public keys, a client typically stores a server's public key locally the first time it enters into an SSH session with that server; in subsequent sessions, it uses this to verify signed messages received from the server. We prove the end-to-end property of a secure channel, which relies on the network protocol being secure *assuming* the client's copy of the server's key is integrity protected, and, in addition, the client's copy of the server's key *actually* being integrity protected on her local machine.

Extensions. In ongoing work, we are using LS^2 to verify protocols and applications based on hardware for trusted computing [26]. The axioms for reasoning about shared memory and time presented in this paper appear to be general enough for this application. We have successfully extended LS^2 to model trusted computing primitives such as Platform Configuration Registers [26], as well as many cryptographic primitives like encryption and hashing without much difficulty. We have also modeled extremely strong adversaries who may reset running machines at arbitrary times. We are currently considering extensions of LS^2 with primitives that allow branching to arbitrary code in memory whose contents may not be known in advance. This situation is relevant because attestation protocols for trusted computing are expressly designed to prove the integrity of the software stack even in the presence of adversaries who can modify code. This has turned out to be a technically challenging problem. We also plan to apply LS^2 to analysis of virtual machines and hypervisors [1, 27]. For these applications, we expect a formalism that will restrict the interfaces available to different programs. For example, a process running inside a virtual machine may not be able to read and write memory except through special hooks that are provided by the virtual machine. 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. We also plan to develop tool support for mechanized reasoning. The broader research goal of this effort is to bring the same kind of rigor to the study of secure systems that current methods and tools do successfully for security protocols.

Related Work. LS^2 shares a number of features with logics of programs [5, 7, 12, 13, 16, 19]. As mentioned earlier, one central difference from PCL is that LS^2 considers shared memory systems in addition to network communication. Although concurrent separation logic [5] 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 [15, 16, 19] 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 adversary. 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.

The rest of the paper is organized as follows. Section 2 informally describes our central modeling choices. Section 3 describes a programming language in which secure systems are specified. Section 4 describes the syntax of the logic LS^2, its semantics, proof system, and soundness theorem. It also illustrates use of the logic by applying it to the example protocol. Section 5 concludes the paper. Proofs are available in the full version of the paper, which is available from the authors' homepages.

2 Overview of Modeling Choices

A secure system is specified as a set of programs that represent relevant parts of the system. For example, a typical two party client-server protocol will contain two programs, one to be executed by the client and the other by the server. We describe below at a high level two of our core modeling choices. The technical details of the programming language reflecting these choices are described in the next section.

Memory locations and protection. We assume a set of machines, and on each machine we assume an abstract set of locations of memory. These locations model both the RAM, and persistent store such as hard disks. Locations may be read and written using special actions in the programming language. We also model an abstract form of memory protection through locking. A running thread may obtain an exclusive-write lock on any location of memory on the machine on which it is executing. An exclusive-write lock means that only the thread holding the lock may write the location; other threads may, however, read it. An unlocked location on a machine may be written by any thread on the machine, including possibly a malicious thread. Although we do not do so in this paper, it should be easy to add exclusive-read-write locks that preclude all other threads from reading and writing the locked location. Such locks may be necessary if we wish to reason about secrecy properties. The use of locks in our formalism differs significantly from their use in logics for reasoning about concurrent programs [5], where locks are used to guarantee mutual exclusion in critical regions of code. The two notions are equally expressive, but locks on memory locations seem more suited to modeling secure systems.

Adversary Model. We formally model adversaries as extra threads executing concurrently with protocol participants. The *local adversary* threads may be located on any machine, and may execute any program. Such threads can, for example, use any of the following capabilities to attack the system: 1) reading any memory location and writing to any memory location that is not explicitly locked by other threads, 2) sending the contents of memory over the network, and 3) acquiring a lock on any unlocked location. Consequently, the adversary is powerful enough to exploit race conditions and launch TOCTTOU attacks. The *network adversary* is modeled in the standard way as in prior work [7]: the adversary can remove any message from the network and send

Expressions	e	::=	n	Number
		|	$\hat{X}, \hat{Y}$	Agent
		|	K	Key
		|	K^{-1}	Inverse of key K
		|	x	Variable
		|	(e, e')	Pair
		|	$SIG_K\{\!\|e\|\!\}$	Value e signed by key K
Machine	m			
Location	l			
Action	a	::=	`read` l	Read location l
		|	`write` l, e	Write e to location l
		|	`send` e	Send e as a message
		|	`receive`	Receive a message
		|	`sign` e, K^{-1}	Sign e with private key K^{-1}
		|	`verify` e, K	Check that $e = SIG_{K^{-1}}\{\!\|e'\|\!\}$
		|	`lock` l	Obtain write lock on location l
		|	`unlock` l	Release write lock on location l
		|	$\texttt{proj}_1\ e$	Project the 1st component of a pair
		|	$\texttt{proj}_2\ e$	Project the 2nd component of a pair
		|	`match` e, e'	Check that $e = e'$
		|	`new`	Generate a new nonce
Program	P, Q	::=	$x_1 := a_1; \ldots; x_n := a_n$	
Thread id	I	::=	$\langle \hat{X}, \eta, m \rangle$	
Thread identifier	η			
Thread	T, S	::=	$[P]_I$	
Store	σ	:	Locations $\rightarrow$ Expressions	
Lock map	ι	:	Locations $\rightarrow$ (Thread ids) $\cup$ $\{\_\}$	
Configuration	C	::=	$\iota, \sigma, T_1 \mid \ldots \mid T_n$	

Figure 1: Syntax of the programming language

messages that it can create following the symbolic attacker model [9, 18]. The local and network adversaries can communicate with each other by sending messages.

3 A Programming Language for Specifying Systems

Our language for specifying systems descends from the corresponding language in PCL and extends the latter with constructs for reading, writing, and protecting memory. Its syntax is summarized in Figure 1. We assume an algebra of expressions (denoted e). Expressions may be numbers n, identities of agents $\hat{X}$, keys K, or variables x. We allow pairing of expressions, written (e, e'), and signatures using private keys: $SIG_K\{\!|e|\!\}$ denotes the signature on e made using the key K. We assume that the expression e may be recovered from the signature if the verification key corresponding to K is known. Expressions are assumed to be typed (for e.g., a pair can be distinguished from a number), but we elide the details of the types. The algebra may be extended with constructs for other cryptographic operations like encryption and hashing if required.

Agents and keys. Agents, denoted $\hat{X}, \hat{Y}$, are principals associated with a system (e.g., users). Keys are denoted by the letter K. The inverse of key K is denoted by K^{-1}. As a convention, we use the notation K^{-1} for private keys and the notation K for public keys. If K is a private key, we write $\hat{K}$ to denote the agent who owns it. If

K is a public key, we write $\hat{K}$ to denote the agent who owns the corresponding private key. By definition, $\hat{K} = \hat{K^{-1}}$.

Machines and locations. Machines (denoted m) are the sites of program execution, and the sites that hold locations of memory. Locations, l, model both RAM and persistent store such as disks. The function $machine(l)$ returns the machine on which location l exists. For clarity, we often prefix the name of a location with the name of the machine on which it exists, writing $m.l$ instead of l if $machine(l) = m$.

Actions and programs. Actions, denoted a, perform specific functions such as reading or writing a location of memory, sending or receiving a message, creating or verifying a signature, obtaining or releasing a lock on a location of memory, projecting the components of a pair, checking that two expressions are equal, or generating a new nonce. Allowed actions with their intuitive meanings are listed in Figure 1. The actions `send` and `receive` are undirected; an expression sent by any program may be received by any other program.

Actions may not always succeed: a signature verification may fail, for instance. If an action fails, we assume that the thread executing the action blocks forever. A successfully executed action always returns an expression. For example, the action `receive` returns the expression obtained by synchronizing with another thread, and the action `verify` returns the message contained in the signature it verifies. The expressions returned by the actions `write`, `send`, `lock`, `unlock`, and `match` are unimportant. We assume that these actions always return the constant 0.

A program (denoted P, Q) is a sequence $x_1 := a_1; \ldots; x_n := a_n$ of actions. The number n may be zero, in which case the program is empty. The variable x_i binds to the value returned by the action a_i. The scope of each variable extends to the end of the program. Variables may be α-varied. In this sense, our treatment of variables is functional, not imperative. (Imperative stores are modeled by locations.) We often write $P(e/x)$ to mean the program P with e substituted for x. This substitution is capture avoiding, as usual. We also write $P; Q$ to mean the program obtained by concatenating the actions in P and Q. The scope of variables bound in P extends over Q.

Threads and Thread ids. A thread T is a sequentially executing program. Formally, it is a pair containing a program and an identity I, written $[P]_I$. The identity (id) I is a three tuple $\langle \hat{X}, \eta, m \rangle$. $\hat{X}$ is the agent who owns the thread, η is a unique identifier for the thread (akin to a process id), and m is the machine on which the thread executes. For $I = \langle \hat{X}, \eta, m \rangle$, we define $\hat{I} = \hat{X}$ and $machine(I) = m$.

Configurations. A configuration C is the collection of all threads executing concurrently on all machines. Concurrent threads are separated by the symbol $|$, which is assumed to be commutative and associative. In addition to threads, a configuration also contains a *store* σ, which is a map from the set of all locations to the values that they contain, and a *lock map* ι that maps each location to the id of the thread that has an exclusive-write lock on it. If no thread has a lock on location l, then $\iota(l) = \_$. We often write $\sigma[l \mapsto e]$ to denote the map σ augmented with the mapping of l to e. $\iota[l \mapsto I]$ is defined similarly. We assume implicitly that all threads in a configuration are closed, i.e., they do not contain any free variables.

$$
\begin{array}{lll}
\text{(sign)} & [x := \texttt{sign}\, e, K^{-1};\ P]_I \longrightarrow [P(SIG_{K^{-1}}\{|e|\}/x)]_I & \\
\text{(verify)} & [x := \texttt{verify}\, SIG_{K^{-1}}\{|e|\}, K;\ P]_I \longrightarrow [P(e/x)]_I & \\
\text{(proj1)} & [x := \texttt{proj}_1\, (e_1, e_2);\ P]_I \longrightarrow [P(e_1/x)]_I & \\
\text{(proj2)} & [x := \texttt{proj}_2\, (e_1, e_2);\ P]_I \longrightarrow [P(e_2/x)]_I & \\
\text{(match)} & [x := \texttt{match}\, e, e;\ P]_I \longrightarrow [P(0/x)]_I & \\
\text{(new)} & [x := \texttt{new}\,;\ P]_I \longrightarrow [P(n/x)]_I & (n \text{ fresh}) \\
\text{(read}^*\text{)} & \sigma, [x := \texttt{read}\, l;\ P]_I \longrightarrow \sigma, [P(e/x)]_I & (\sigma(l) = e) \\
\text{(write}^*\text{)} & \iota, \sigma[l \mapsto e'], [x := \texttt{write}\, l, e;\ P]_I \longrightarrow \iota, \sigma[l \mapsto e], [P(0/x)]_I & (\iota(I) \in \{I, \_\}) \\
\text{(lock}^*\text{)} & \iota[l \mapsto \_], [x := \texttt{lock}\, l;\ P]_I \longrightarrow \iota[l \mapsto I], [P(0/x)]_I & \\
\text{(unlock}^*\text{)} & \iota[l \mapsto I], [x := \texttt{unlock}\, l;\ P]_I \longrightarrow \iota[l \mapsto \_], [P(0/x)]_I & \\
\text{(comm)} & [x := \texttt{send}\, e;\ P]_I \mid [y := \texttt{receive}\,;\ Q]_{I'} \longrightarrow [P(0/x)]_I \mid [Q(e/y)]_{I'} &
\end{array}
$$

* Side Condition: $machine(l) = machine(I)$

Figure 2: Reduction Rules of the Process Calculus

3.1 Reduction Rules and Traces

The operational semantics of the programming language are defined by reduction rules on configurations, summarized in Figure 2. In each rule, we include only the relevant parts of configurations. Parts not shown in a rule remain unchanged in the reduction. Rules (sign)–(new) represent internal reductions of a thread. In the rule (new), the value n is a fresh symbolic constant that has never occurred in the history of the configuration. This rule is used for generating nonces. The rules (read) and (write) allow reading and writing locations. In each case, the location read or written must be on the same machine as the thread executing the action. In the case of writing, the location being written must either be locked by the thread executing the action, or be unlocked (enforced by the side condition $\iota(l) \in \{I, \_\}$). Rules (lock) and (unlock) allow a thread to obtain and release an exclusive-write lock on a location. This change is noted in the lock map ι. Rule (comm) allows communication between any two threads even if they are on different machines.

Traces and Timed Traces. A trace is a sequence of configurations $C_0 \longrightarrow \ldots \longrightarrow C_n$ such that C_{i+1} may be obtained from C_i by one of the reduction rules. Given a configuration C_0, there may be many possible traces starting from it because internal actions of different threads may be interleaved arbitrarily and also because communication is non-deterministic. A *timed trace* (denoted $\mathcal{T}$) is a trace in which a real number has been associated with each reduction. We write a timed trace as $C_0 \xrightarrow{t_1} C_1 \ldots \xrightarrow{t_n} C_n$. The real numbers $t_1, \ldots, t_n$ represent points of time at which the reductions happened. We require that $t_1 < \ldots < t_n$, i.e., the time points be monotonically increasing. It is assumed that the effects of a reduction, such as changes to the store or the lock map, come into effect immediately *after* the time at which the reduction happens, but not at the time of reduction itself.

3.2 Example: An Extended Challenge Response Protocol

As an illustrative example, we consider an extension of a standard challenge response protocol for authenticating a server to a client with explicit steps for storing and fetching the server's public key in a file on the client side. An exclusive-write lock on the file guarantees the integrity of the stored key during the protocol. This idea of storing

$Client(m, K_S, \hat{C}) \equiv$
`lock` $m.pk$;
`write` $m.pk, K_S$;
——
$n :=$ `new` ;
`send` $(\hat{C}, n)$;
$(s, r) :=$ `receive` ;
`match` $s, \hat{K_S}$;
$k :=$ `read` $m.pk$;
$(n', c) :=$ `verify` r, k;
`match` n', n;
`match` $c, \hat{C}$;
`unlock` $m.pk$

$Server(m', K_S^{-1}) \equiv$
$(c, n) :=$ `receive` ;
$r :=$ `sign` $(n, c), K_S^{-1}$;
`send` $(\hat{K_S^{-1}}, r)$

Figure 3: An Extended Challenge Response Protocol

the server's public key in a file is motivated by the SSH protocol [28], where clients routinely store public keys of servers in a "known hosts" file. The programs of both the client and the server are listed in Figure 3. We assume that the client agent is $\hat{C}$, and that the public key of the server is K_S. For actions such as `lock` , `unlock` , and `write` that do not produce a useful result, we omit the binding variable. $(x, y) := a$ is an abbreviation for $z := a; x :=$ `proj`$_1$ $z; y :=$ `proj`$_2$ z. Both the client's and the server's programs contain several parameters, including the machines m and m' on which they execute.

The client's program contains two parts, separated by —— for readability. In the first part (above ——), the client obtains an exclusive write lock on the location $m.pk$, which models a file on the disk, and stores the server's public key K_S in it. This step may be executed well in advance of the remaining protocol. In the second part, the client generates a nonce n and sends it to the server, together with its own identity $\hat{C}$. Then it receives a signed message r from the server together with the server's id s. It verifies that s matches $\hat{K_S}$ and also that r contains the nonce n and its own id $\hat{C}$. In order to verify the signature in r, *the client reads the public key of server* from the location $m.pk$, where it had stored the key earlier. The lock on $m.pk$ obtained in the first part ensures that the key was not changed in the meantime by another thread.

The server's program is straightforward. It receives the nonce n and the client's id c, produces a signature r on both with its private key, and sends the signature together with its own id to the client. In later sections we use the logic LS^2 to show formally that this protocol is correct, in the sense that the client is guaranteed that it was communicating with the intended server. The lock on $m.pk$ is central to this correctness.

4 LS^2: Syntax, Semantics, and Proof System

Formulas of LS^2 express properties of timed traces obtainable from programs. LS^2 descends from PCL, and like its predecessor it is an extension of first-order logic. Besides some predicates for reasoning about memory locations and locks, the main difference between PCL and LS^2 is the presence of explicit time in the latter. It is possible to say in LS^2 that a formula A is true at exactly time t, written $A \ @ \ t$. For example, the fact that location l contains the constant 0 at time t may be written $\mathsf{Mem}(l, 0) \ @ \ t$. Explicit time is also used for ordering events during reasoning.

Action Predicates	R	::=	$\mathsf{Read}(I, l, e)$	`read` l getting value e
		|	$\mathsf{Write}(I, l, e)$	`write` l, e
		|	$\mathsf{Send}(I, e)$	`send` e
		|	$\mathsf{Receive}(I, e)$	`receive` receiving e
		|	$\mathsf{Sign}(I, e, K)$	`sign` e, K
		|	$\mathsf{Verify}(I, e, K)$	`verify` $SIG_{K^{-1}}\{\lvert e \rvert\}, K$
		|	$\mathsf{Lock}(I, l)$	`lock` l
		|	$\mathsf{Unlock}(I, l)$	`unlock` l
		|	$\mathsf{Match}(I, e, e')$	`match` e, e'
		|	$\mathsf{New}(I, n)$	`new` generating nonce n
General Predicates	M	::=	$\mathsf{Mem}(l, e)$	Location l contained e
		|	$\mathsf{IsLocked}(l, I)$	Thread I had a lock on l
		|	$\mathsf{Contains}(e, e')$	
		|	$e = e' \mid t \geq t'$	
		|	$\mathsf{Honest}(\hat{X})$	
		|	$\mathsf{Honest}(\hat{X}, \vec{P})$	
Formulas	A, B	::=	$R \mid M \mid \top \mid \bot \mid A \wedge B \mid A \vee B \mid$ $A \supset B \mid \neg A \mid \forall x.A \mid \exists x.A \mid A \mathbin{@} t$	
Defined Formula	A on i	=	$\forall t.\, ((t \in i) \supset (A \mathbin{@} t))$	
Modal Formulas	J	::=	$[P]_I^{t_b,t_e}\ A \quad \mid \quad [a]_{I,x}^{t_b,t_e}\ A$	

Figure 4: Syntax of LS^2

4.1 Syntax

The syntax of formulas is shown in Figure 4. We classify predicates into two categories. *Action predicates* express that certain actions happened on a timed trace. For example, the predicate $\mathsf{New}(I, n)$ holds on a trace $\mathcal{T}$ at time t if and only if a thread with id I executed action `new` at time t in $\mathcal{T}$, and received the nonce n as a result of the action. The actions corresponding to each action predicate are listed in the figure. *General predicates* capture those properties of a trace that are not specific to actions. The predicate $\mathsf{Mem}(l, e)$ holds at time t on a trace, if location l contained e at time t on the trace. Similarly, $\mathsf{IsLocked}(l, I)$ holds on a trace at time t iff thread I had an exclusive-write lock on location l at time t. $e = e'$ represents equality of expressions, while $\mathsf{Contains}(e, e')$ means that e contains e' as a sub-expression.

Time points, denoted t, are elements of any totally ordered, dense set[1] with a maximum element ∞ and a minimum element $-\infty$. For simplicity, we assume that this set is $\mathbb{R} \cup \{-\infty, \infty\}$, ordered with the usual total order $\geq$ on real numbers extended with $-\infty$ and ∞. Other comparison operators for time points $\leq, >, <, \neq$ may be defined using equality and $\geq$ as usual. Density of time points is needed to prove soundness of one of the rules of the proof system (rule (Seq) in Figure 5), but is not axiomatized in the proof system.

Honesty. In the analysis of systems it is often necessary to assume that some agents are not adversarial, and execute only the programs specified. For example, in the protocol of Section 3.2, if the server is adversarial, it may leak its private key and thus compromise the authentication guarantee provided by the protocol. To this end, we sometimes designate an agent $\hat{X}$ as being *honest*, written as the predicate $\mathsf{Honest}(\hat{X})$. $\mathsf{Honest}(\hat{X})$ means that no thread of $\hat{X}$ ever leaks $\hat{X}$'s private keys by sending them

[1] By dense we mean that between any two non-identical time points there exists a third time point.

in a message, or by writing them to a memory location. In addition, we sometimes assume that a given honest agent executes only fixed programs. For example, it may be reasonable to assume that the server $\hat{K_S}$ in the CR protocol only executes the role $Server(m', K_S^{-1})$ from Figure 3, and never initiates the protocol as client. To represent such assumptions, we use the predicate $\mathsf{Honest}(\hat{X}, \vec{P})$. This predicate implies the predicate $\mathsf{Honest}(\hat{X})$ and in addition means that threads owned by $\hat{X}$ execute only the programs in the set $\vec{P}$. The set of honest agents is assumed to be known in advance, and is associated implicitly with a (timed) trace.

Formulas. Formulas, denoted A, may be predicates, or they may be constructed using the usual connectives of first-order logic. Quantifiers may range over expressions, thread ids, time points, and locations. We also allow the formula $A \ @ \ t$ meaning that formula A is true at time t. Quite often in proofs we need to say that a formula A holds throughout the interval i, where i has one of the forms (t_1, t_2), $[t_1, t_2)$, $(t_1, t_2]$, or $[t_1, t_2]$. To represent these, we define the shorthand notation $(A \text{ on } i)$ as $\forall t.\ ((t \in i) \supset (A \ @ \ t))$. The membership predicate $t \in i$ is defined in the obvious way for each type of interval. For example, if $i = [t_1, t_2]$, then $t \in i = (t_1 \leq t) \wedge (t_2 \geq t)$.

In addition to the usual formulas A, LS^2 includes two types of *modal formulas* for reasoning about programs. The formula $[P]_I^{t_b,t_e} A$ means that if the thread with id I executes all actions in P in the time interval $[t_b, t_e)$ (and no others), then formula A holds. The related formula $[a]_{I,x}^{t_b,t_e} A$ means that if thread I executes only the action a in the interval $[t_b, t_e)$, returning the result x, then A holds. As a general rule, t_b, t_e, and x are always parameters. The formula A cannot mention variables bound in P. It may however, mention t_b, t_e, and any variables free in P or in a. In the modal formula $[a]_{I,x}^{t_b,t_e} A$, A may mention x also. This allows us to incorporate the result of executing an action into logical reasoning.

4.1.1 Example of a Correctness Property

Consider the following correctness property for the challenge response protocol in Section 3.2.

If a thread I executes the program $Client(m, K_S, \hat{C})$ successfully, then at some time t_g the thread I generated a nonce n, at some later time t_r, a thread I' owned by $\hat{K_S}$ received n in a message, and later, at some time t_s, I' created a signature on the pair $(n, \hat{C})$.

Let us assume that the client program executes in a thread with id I. Then the correctness property above may be expressed in LS^2 as the modal formula:

$$
\begin{aligned}
J_{CR} \ = \ & [Client(m, K_S, \hat{C})]_I^{t_b,t_e} \quad \exists n.\exists t_g.\exists t_r.\exists t_s.\exists I'.\ ((t_b < t_g < t_r < t_s < t_e) \\
& \wedge (\mathsf{New}(I, n) \ @ \ t_g) \wedge (\hat{I'} = \hat{K_S}) \wedge \\
& (\mathsf{Receive}(I', (\hat{C}, n)) \ @ \ t_r) \wedge \\
& (\mathsf{Sign}(I', (n, \hat{C}), K_S^{-1}) \ @ \ t_s))
\end{aligned}
$$

The modal annotation $[Client(m, K_S, \hat{C})]_I^{t_b,t_e}$ means that the remaining formula holds whenever thread I successfully executes the program $Client(m, K_S, \hat{C})$ in the interval $[t_b, t_e)$. The formula after this modal annotation states the correctness property. The @ connective is used to specify the time at which actions occurred. The actions are ordered using the natural order on time points ($t_g < t_r < t_s$).

The above property cannot be established without assuming that $\hat{K_S}$ is honest, i.e., it does not leak its private key, and that it executes the role of the server. We also

need to assume that $\hat{I} \neq \hat{K_S}$. We use the notation Γ_{CR} to denote these assumptions. $\Gamma_{CR} = \{\mathsf{Honest}(\hat{K_S}, Server(m', K_S^{-1})), \hat{I} \neq \hat{K_S}\}$. In Section 4.5 we show formally that Γ_{CR} entails J_{CR} in LS^2.

4.2 Semantics

The formulas of LS^2 are interpreted over timed traces. The basic semantic judgment, written $\mathcal{T} \models^t A$, means that A holds in the trace $\mathcal{T}$ at time t. It is defined by induction on the structure of A.

Action Predicates. If A is an action predicate, $\mathcal{T} \models^t A$ holds if a reduction corresponding to A occurred at time t in $\mathcal{T}$. For example,

$\mathcal{T} \models^t \mathsf{Read}(I, l, e)$ if thread I executed action `read` l at time t, reading e from location l.

$\mathcal{T} \models^t \mathsf{Lock}(I, l)$ if thread I executed action `lock` l at time t.

General Predicates.

$\mathcal{T} \models^t \mathsf{Mem}(l, e)$ if the location l contained e at time t, i.e., at time t, $\sigma(l) = e$.

$\mathcal{T} \models^t \mathsf{IsLocked}(l, I)$ if at time t, $\iota(l) = I$.

$\mathcal{T} \models^t \mathsf{Contains}(e, e')$ if e' is a subexpression of e.

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

$\mathcal{T} \models^t t_1 \geq t_2$ if $t_1 \geq t_2$ in the usual ordering of real numbers.

$\mathcal{T} \models^t \mathsf{Honest}(\hat{X})$ if it is assumed that $\hat{X}$ does not leak its private key.

$\mathcal{T} \models^t \mathsf{Honest}(\hat{X}, \vec{P})$ if $\mathcal{T} \models^t \mathsf{Honest}(\hat{X})$, and all threads in $\mathcal{T}$ owned by $\hat{X}$ execute programs in $\vec{P}$ only.

Formulas. Formulas are interpreted in a standard way. The only new case is that for $A \mathbin{@} t$.

$\mathcal{T} \models^t \top$.

$\mathcal{T} \not\models^t \bot$.

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

$\mathcal{T} \models^t A \vee B$ if $\mathcal{T} \models^t A$ or $\mathcal{T} \models^t B$.

$\mathcal{T} \models^t A \supset B$ if $\mathcal{T} \not\models^t A$ or $\mathcal{T} \models^t B$.

$\mathcal{T} \models^t \neg A$ if $\mathcal{T} \not\models^t A$.

$\mathcal{T} \models^t \forall x.A$ if for each ground instance v of x, $\mathcal{T} \models^t A(v/x)$.

$\mathcal{T} \models^t \exists x.A$ if there exists a ground instance v of x such that $\mathcal{T} \models^t A(v/x)$.

$\mathcal{T} \models^t A \mathbin{@} t'$ if $\mathcal{T} \models^{t'} A$.

It should be observed that by definition the relation $\mathcal{T} \models^t A \mathbin{@} t'$ is independent of t. This is consistent with semantics of other hybrid logics (e.g., [4]).

Modal Formulas. For modal formulas, the semantic judgments are written $\mathcal{T} \models [P]_I^{t_b,t_e} A$ and $\mathcal{T} \models [a]_{I,x}^{t_b,t_e} A$. As opposed to the judgment $\mathcal{T} \models^t A$, these judgments are not relativized to time because modal formulas express properties of programs and actions, and are independent of time. Intuitively, $\mathcal{T} \models [P]_I^{t_b,t_e} A$ holds if in the trace $\mathcal{T}$, either the thread with id I does not execute the sequence of actions P in the interval $[t_b, t_e)$ *or* A holds. To state this formally, we define a notion of matching between a timed trace $\mathcal{T}$ and a modal prefix $[P]_I^{t_b,t_e}$.

Matching. We say that a timed trace $\mathcal{T} = C_0 \xrightarrow{t_1} C_1 \ldots \xrightarrow{t_n} C_n$ matches the modal prefix $[P]_I^{t_b,t_e}$ with substitution θ, written $\mathcal{T} \gg [P]_I^{t_b,t_e} \mid \theta$ if the following hold:

1. For some program Q, some configuration C_i contains the thread $[(P; Q)\theta]_I$, i.e., a substitution instance of the program $P; Q$ running under id I.
2. For some $i' \geq i$ and some substitution φ, $C_{i'}$ contains $[Q\varphi]_I$.
3. The time associated with each reduction of I between C_i and $C_{i'}$ lies in the interval $[t_b, t_e)$.

If P does not contain any actions, we trivially define $\mathcal{T} \gg [P]_I^{t_b,t_e} \mid \theta$ for any substitution θ if in $\mathcal{T}$ thread I does not perform any reduction in the interval $[t_b, t_e)$.

In a similar manner, for an action a, we define $\mathcal{T} \gg [a]_{I,x}^{t_b,t_e} \mid \theta, e/x$ if $\mathcal{T} \gg [x := a]_I^{t_b,t_e} \mid \theta$, when $x := a$ is viewed as a program with only one action, and in addition the execution of action a in $\mathcal{T}$ produces the result e.

Given this definition of matching, we define semantic satisfaction for modal formulas as follows.

$\mathcal{T} \models [P]_I^{t_b,t_e} A$ if for each θ, and all ground time points t, t'_b and t'_e, $\mathcal{T} \gg [P]_I^{t'_b,t'_e} \mid \theta$ implies $\mathcal{T} \models^t A\theta(t'_b/t_b)(t'_e/t_e)$.

$\mathcal{T} \models [a]_{I,x}^{t_b,t_e} A$ if for each θ, all ground time points t, t'_b and t'_e, and each ground e, $\mathcal{T} \gg [a]_{I,x}^{t'_b,t'_e} \mid \theta, e/x$ implies $\mathcal{T} \models^t A\theta(t'_b/t_b)(t'_e/t_e)(e/x)$.

4.3 Proof System for LS^2

Correctness properties of systems such as the one in Section 4.1.1 are proved using a proof system for LS^2, which we describe now. A soundness theorem (Section 4.4), shows that any property so established actually holds on all traces. If Φ is a formula (modal or otherwise), we write $\vdash \Phi$ to mean that Φ is provable using the proof system. The rules and axioms of the proof system are shown in Figures 5 and 6. In addition to these rules and axioms, we assume a full axiomatization of first-order logic, and axioms that make the set of time points a total order. We also assume that equality of expressions is an equivalence relation. Further, we assume some axioms for the predicate $\mathsf{Contains}(e, e')$, often relying on types of terms (for e.g., if e is a number, then $\mathsf{Contains}(e, e') \supset e = e'$). These straightforward axioms are elided here.

Rule (NecAt) states that if A is provable, then so is $A \mathbin{@} t$. This rule is akin to the so called "necessitation" rule from standard modal logics (see e.g., [11]). The basic rule used for reasoning about modal formulas is (Seq). If we know that the program $x := a; P$ was executed in the interval $[t_b, t_e)$, then by the density of time points, there must be a time point t_m such that action a executed in the interval $[t_b, t_m)$ and P

$$\frac{\vdash A}{\vdash A \mathbin{@} t}\text{NecAt} \qquad \frac{\vdash [a]^{t_b,t_m}_{I,x}\ A_1 \qquad \vdash [P]^{t_m,t_e}_{I}\ A_2 \qquad (t_m \text{ fresh})}{\vdash [x := a; P]^{t_b,t_e}_{I}\ \exists t_m.\exists x.\ ((t_b < t_m < t_e) \wedge A_1 \wedge A_2)}\text{Seq}$$

$$\frac{\vdash [P]^{t_b,t_e}_{I}\ A_1 \qquad \vdash [P]^{t_b,t_e}_{I}\ A_2}{\vdash [P]^{t_b,t_e}_{I}\ A_1 \wedge A_2}\text{Conj1} \qquad \frac{\vdash [a]^{t_b,t_e}_{I,x}\ A_1 \qquad \vdash [a]^{t_b,t_e}_{I,x}\ A_2}{\vdash [a]^{t_b,t_e}_{I,x}\ A_1 \wedge A_2}\text{Conj2}$$

$$\frac{\vdash [P]^{t_b,t_e}_{I}\ A_1 \supset A_2 \qquad \vdash [P]^{t_b,t_e}_{I}\ A_1}{\vdash [P]^{t_b,t_e}_{I}\ A_2}\text{Imp1} \qquad \frac{\vdash [a]^{t_b,t_e}_{I,x}\ A_1 \supset A_2 \qquad \vdash [a]^{t_b,t_e}_{I,x}\ A_1}{\vdash [a]^{t_b,t_e}_{I,x}\ A_2}\text{Imp2}$$

$$\frac{\vdash A}{\vdash [P]^{t_b,t_e}_{I}\ A}\text{Nec1} \qquad \frac{\vdash A}{\vdash [a]^{t_b,t_e}_{I,x}\ A}\text{Nec2} \qquad \frac{\forall Q \in IS(\vec{P}).\ \vdash [Q]^{t_b,t_e}_{I}\ A}{\vdash \mathsf{Honest}(\hat{I},\vec{P}) \supset \forall t_e.\ A(-\infty/t_b)}\text{Honesty}$$

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

reduced in the interval $[t_m, t_e)$. Using this fact, we may reason about the action $x := a$ and the program P in isolation and combine the two properties. The side condition (t_m fresh) means that t_m should not appear free in A_1 and A_2 and that it should be distinct from both t_b and t_e. Rules (Conj1) – (Nec2) allow us to incorporate reasoning about ordinary formulas into modal formulas.

$IS(\vec{P})$ in the rule (Honesty) denotes programs that are prefixes of programs in the set $\vec{P}$. Formally, $IS(\vec{P}) = \{x_1 := a_1; \ldots; x_i := a_i \mid (x_1 := a_1; \ldots; x_n := a_n) \in \vec{P} \text{ and } 0 \leq i \leq n\}$. $IS(\vec{P})$ also includes the empty program. The rule (Honesty) may be interpreted as follows: if we know that thread I is executing one of the programs in the set $\vec{P}$ (assumption $\mathsf{Honest}(\hat{I}, \vec{P})$), and on all prefixes of programs in this set some property A holds (premise), then property A must hold.

Figure 6 describes the axioms of LS^2. Axioms (K) and (Disj) state basic properties of the connective @. These axioms, together with the rule (NecAt) imply that $(A \wedge B) \mathbin{@} t \equiv (A \mathbin{@} t) \wedge (B \mathbin{@} t)$ and that $(A \vee B) \mathbin{@} t \equiv (A \mathbin{@} t) \vee (B \mathbin{@} t)$, where $A \equiv B$ denotes logical equivalence defined as $(A \supset B) \wedge (B \supset A)$. In axioms (Act)–(¬Act), we use the notation $R(I, x, a)$ as an abbreviation for the action predicate corresponding to the action $x := a$, performed by thread with id I. For example, if $a = \texttt{receive}$, then $R(I, x, a) = \mathsf{Receive}(I, x)$, and if $a = \texttt{send}\ e$, then $R(I, x, a) = \mathsf{Send}(I, e)$. As a syntactic convention, we assume that ¬ binds tighter than on. The axiom (Act) states that if thread I executes only the action a in the interval $[t_b, t_e)$ returning value x, then there is a time point t in the interval such that $R(I, x, a)$ holds at time t, and that $R(I, x, a)$ does not hold at any other point interval $[t_b, t_e)$.

Axioms (Verify)–(Write) describe properties of specific actions. Of particular importance are axioms (Lock) and (Write). (Lock) states that if thread I executes only the action $\texttt{lock}\ l$ in the interval $[t_b, t_e)$, then at time t_e, I has a lock on l. (Write) states that if a thread I writes expression e to location l in some interval, then there is at least one time point t in the interval when the location contained the expression, and that I does not write to l after that time point during the interval.

Axiom (Mem=) states that the same location cannot contain two different expressions at the same time. (MemI) is an invariance axiom: if thread I has an exclusive-write lock on location l during an interval, and it does not write to the location in the interval, then the expression contained in the location does not change during the

(K) $\vdash ((A \supset B)\ @\ t) \supset ((A\ @\ t) \supset (B\ @\ t))$

(Disj) $\vdash ((A \vee B)\ @\ t) \supset ((A\ @\ t) \vee (B\ @\ t))$

(Eq) $\vdash ((e = e') \wedge A(e/x)) \supset A(e'/x)$

(Act) $\vdash [a]^{t_b,t_e}_{I,x}\ \exists t.\ t \in [t_b, t_e) \wedge (R(I,x,a)\ @\ t) \wedge (\neg R(I,x,a)\ \mathsf{on}\ [t_b, t)) \wedge (\neg R(I,x,a)\ \mathsf{on}\ (t, t_e))$

(Act') $\vdash [\,]^{t_b,t_e}_{I}\ \neg R(I,x,a)\ \mathsf{on}\ [t_b, t_e)$

(¬Act) $\vdash [a]^{t_b,t_e}_{I,x}\ \neg R(I,x',a')\ \mathsf{on}\ [t_b, t_e)$ if $x \neq x'$ or $a \neq a'$

(Verify) $\vdash [\mathtt{verify}\ e, K]^{t_b,t_e}_{I,x}\ e = SIG_{K^{-1}}\{\!|x|\!\}$

(Sign) $\vdash [\mathtt{sign}\ e, K^{-1}]^{t_b,t_e}_{I,x}\ x = SIG_{K^{-1}}\{\!|e|\!\}$

(Proj1) $\vdash [\mathtt{proj}_1\ e]^{t_b,t_e}_{I,x}\ \exists e'.\ e = (x, e')$

(Proj2) $\vdash [\mathtt{proj}_2\ e]^{t_b,t_e}_{I,x}\ \exists e'.\ e = (e', x)$

(Match) $\vdash (\mathsf{Match}(I, e, e')\ @\ t) \supset e = e'$

(Lock) $\vdash [\mathtt{lock}\ l]^{t_b,t_e}_{I,x}\ \mathsf{IsLocked}(l, I)\ @\ t_e$

(Write) $\vdash [\mathtt{write}\ l, e]^{t_b,t_e}_{I,x}\ \exists t.\ t \in (t_b, t_e) \wedge (\mathsf{Mem}(l, e)\ @\ t) \wedge ((\forall e'.\ \neg\mathsf{Write}(I, l, e'))\ \mathsf{on}\ [t, t_e))$

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

(MemI) $\vdash (\mathsf{IsLocked}(l, I)\ \mathsf{on}\ [t_b, t_e) \wedge (\mathsf{Mem}(l, e)\ @\ t_b) \wedge (\forall e'.\ \neg\mathsf{Write}(I, l, e')\ \mathsf{on}\ [t_b, t_e))) \supset (\mathsf{Mem}(l, e)\ \mathsf{on}\ [t_b, t_e))$

(LockI) $\vdash ((\mathsf{IsLocked}(l, I)\ @\ t) \wedge (\neg\mathsf{Unlock}(I, l)\ \mathsf{on}\ [t, t'))) \supset (\mathsf{IsLocked}(l, I)\ \mathsf{on}\ [t, t'])$

(VER) $\vdash ((\mathsf{Verify}(I, e, K)\ @\ t) \wedge (\hat{I} \neq \hat{K}) \wedge \mathsf{Honest}(\hat{K})) \supset (\exists I'.\exists t'.\exists e'.\ (t' < t) \wedge (\hat{I'} = \hat{K}) \wedge \mathsf{Contains}(e', SIG_{K^{-1}}\{\!|e|\!\}) \wedge ((\mathsf{Send}(I', e')\ @\ t') \vee \exists l.\ (\mathsf{Write}(I', l, e')\ @\ t')))$

(NEW) $\vdash ((\mathsf{New}(I, n)\ @\ t) \wedge (\mathsf{Receive}(I', e)\ @\ t') \wedge \mathsf{Contains}(e, n)) \supset (t' > t)$

(READ) $\vdash (\mathsf{Read}(I, l, e)\ @\ t) \supset (\mathsf{Mem}(l, e)\ @\ t)$

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

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

interval. (LockI) is a similar invariance axiom for locks.

Axiom (VER) captures the unforgeability of signatures. If a thread I verifies a signature with public key K, and the owner $\hat{K}$ of the corresponding private key is honest, then some thread I' of $\hat{K}$ must have either sent out the signature in a message in the past, or written the signature to a memory location in the past. Axiom (NEW) captures the freshness of nonces: if thread I generates nonce n, and a thread receives a message with the nonce in it, then the latter must have happened *after* the former.

4.4 Soundness

Our main technical result, a soundness theorem, states that if a formula (modal or otherwise) is provable in LS^2's proof system, then it is satisfied by all timed traces. Let Φ denote an arbitrary, possibly modal formula, and let Γ denote a set of non-modal formulas. We say that $\Gamma \vdash \Phi$, if taking the formulas in Γ as axioms, $\vdash \Phi$ may be derived using the proof system. For non-modal formulas A, we say that $\mathcal{T} \models A$ if for each t, $\mathcal{T} \models^t A$. $\mathcal{T} \models \Gamma$ means that for each $B \in \Gamma$, $\mathcal{T} \models B$.

Theorem 1 (Soundness)**.**

1. *If A is a non-modal formula and $\Gamma \vdash A$, then $\mathcal{T} \models \Gamma$ implies $\mathcal{T} \models A$.*

2. *If J is a modal formula and $\Gamma \vdash J$, then $\mathcal{T} \models \Gamma$ implies $\mathcal{T} \models J$.*

The proof of this theorem proceeds by induction on the derivations given in the proof system. Most cases are straightforward, but for the case of the axiom (VER), we make the following assumption:

In the starting configuration of any trace, expressions signed by honest agents do not exist in threads of other agents, nor in memory locations.

4.5 Example of Proof of Correctness

As an illustration of reasoning in LS^2, we prove the following theorem that establishes the correctness property J_{CR} (Section 4.1.1) for the extended challenge response protocol.

Theorem 2 (Correctness)**.** *For Γ_{CR} and J_{CR} defined in Section 4.1.1, $\Gamma_{CR} \vdash J_{CR}$ using the axioms and rules of LS^2.*

Proof (Outline). Since the client's program executes completely, there must be time points t_L and t_W, $t_L < t_W$ at which the actions `lock` $m.pk$ and `write` $m.pk, K_S$ executed (axiom (Act) and rule (Seq)). Further, at a later point t_V, the action `verify` r, k must have executed, where k was read from location $m.pk$. Since in the interim the client's thread neither released the lock on location $m.pk$, nor wrote to the location, $k = K_S$. This deduction uses axioms (Lock), (Write), (LockI) and (MemI) and relies heavily on the fact that the client's program maintains an exclusive write-lock on the location $m.pk$ throughout its execution.

Since the client successfully verified a signature made by the server's public key, and the server's private key is known only to its own threads (assumption $\mathsf{Honest}(\hat{K_S}, Server(m', K_S^{-1}))$), some thread of the server must have sent the signature in a message or written it to memory (axiom (VER)). By a straightforward analysis of the server's program, we now deduce that in the past the server received the nonce n from the client, and generated the signature. By freshness of nonces (axiom (NEW)), both of these must have happened after the nonce was generated. This provides an order on the events: first the client generated the nonce n, then the server received it, and finally the server signed it. This is essentially what we had to prove. □

The analysis of the server's program in this proof is rather standard (see e.g., [7]). The novel part of the proof is the analysis of the lock on the location $m.pk$ and the deduction that (owing to the lock) the value K_S does not change while the client's program executes. Indeed, this lock is essential. In its absence, there is simple a attack on the protocol: an adversary on the client's machine may change the key stored in $m.pk$ before it is read by the client, thus fooling the client into believing that it completed the protocol with $\hat{K_S}$, when it actually did not. The lock prevents this attack. What our formalism allows is precise modeling of such memory protection, and formal proofs that the protection works as expected.

5 Conclusion

We initiate a program to model and analyze end-to-end security properties of contemporary secure systems that rely on network protocols and memory protection. As a concrete first step, 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. The use of the logic is illustrated using a simple example. In subsequent work, we plan to apply the logic to secure systems of practical relevance including trusted computing systems and virtual machine-based secure architectures.

References

[1] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the Symposium on Operating Systems Principles*, 2003.

[2] M. Bishop. Race conditions, files, and security flaws; or the tortoise and the hare redux. Technical Report CSE-95-8, University of California at Davis, Sep 1995.

[3] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2):131–152, 1996.

[4] Torben Braüner and Valeria de Paiva. Towards constructive hybrid logic. In *Electronic Proceedings of Methods for Modalities 3 (M4M3)*, 2003.

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

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

[7] Anupam Datta, Ante Derek, John C. Mitchell, and Arnab Roy. Protocol Composition Logic (PCL). *Electr. Notes Theor. Comput. Sci.*, 172:311–358, 2007.

[8] Henry DeYoung, Deepak Garg, and Frank Pfenning. An authorization logic with explicit time. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF-21)*, June 2008.

[9] D. Dolev and A. Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 2(29), 1983.

[10] Nancy Durgin, John C. Mitchell, and Dusko Pavlovic. A compositional logic for proving security properties of protocols. *Journal of Computer Security*, 11:677–721, 2003.

[11] E. A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science*. The MIT Press, 1990.

[12] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. Foundations of Computing. MIT Press, 2000.

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

[14] Rick Kennell and Leah H. Jamieson. Establishing the genuinity of remote computer systems. In *Proceedings of the 2003 USENIX Security Symposium*, August 2003.

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

[16] Leslie Lamport and Fred B. Schneider. The "hoare logic" of csp, and all that. *ACM Trans. Program. Lang. Syst.*, 6(2):281–296, 1984.

[17] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990.

[18] R.M. Needham and M.D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[19] Susan S. Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.

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

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

[22] Arnab Roy, Anupam Datta, Ante Derek, John C. Mitchell, and Jean-Pierre Seifert. Secrecy analysis in protocol composition logic. *Formal Logical Methods for System Security and Correctness*, 2008.

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

[24] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy platforms. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, October 2005.

[25] Arvind Seshadri, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. SWATT: Software-based attestation for embedded devices. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2004.

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

[27] VMWare. VMWare Workstation. Available at: http://www.vmware.com/, October 2005.

[28] T. Yolen and C. Lonvick. The secure shell (ssh) transport layer protocol. See http://www.ietf.org/rfc/rfc4253.txt, 2006.