2015 IEEE 28th Computer Security Foundations Symposium

A Logic of Programs with Interface-confined Code

Limin Jia
ECE & INI
Carnegie Mellon University
Pittsburgh, USA
liminjia@cmu.edu

Shayak Sen
CSD
Carnegie Mellon University
Pittsburgh, USA
shayaks@cs.cmu.edu

Deepak Garg
Max Planck Institute for
Software Systems
Germany
dg@mpi-sws.org

Anupam Datta
CSD & ECE
Carnegie Mellon University
Pittsburgh, USA
danupam@cmu.edu

***Abstract*—*Interface-confinement* is a common mechanism that secures untrusted code by executing it inside a *sandbox*. The sandbox limits (confines) the code's interaction with key system resources to a restricted set of interfaces. This practice is seen in web browsers, hypervisors, and other security-critical systems. Motivated by these systems, we present a program logic, called System M, for modeling and proving safety properties of systems that execute adversary-supplied code via interface-confinement. In addition to using computation types to specify effects of computations, System M includes a novel *invariant type* to specify the properties of interface-confined code. The interpretation of invariant type includes terms whose effects satisfy an invariant. We construct a step-indexed model built over traces and prove the soundness of System M relative to the model. System M is the first program logic that allows proofs of safety for programs that *execute* adversary-supplied code without forcing the adversarial code to be available for deep static analysis. System M can be used to model and verify protocols as well as system designs. We demonstrate the reasoning principles of System M by verifying the state integrity property of the design of Memoir, a previously proposed trusted computing system.**

I. Introduction

Software systems such as web browsers and mobile OSes, and hypervisors are designed to provide security properties in the presence of adversaries. These adversaries may execute concurrently with trusted programs and access shared resources. They may also supply code to be executed with the privileges of the trusted system. For example, apps from different origins may execute on the same mobile OS platform; browsers routinely execute third-party JavaScript with full access to the page's content; operating system kernels include untrusted (and often buggy) device drivers; and software platforms that leverage the Trusted Platform Module (TPM) [1] execute programs loaded from an untrusted store and only later verify the integrity of the loaded programs. Despite executing potentially adversarial code, these systems have security-related goals, often *safety properties* over traces [2]. For example, a web page must ensure that an embedded untrusted advertisement cannot access a user's password, and trusted computing mechanisms must ensure that the hash chain of the software stack stored in a PCR can only be extended or reset to a fixed value.

One of the common mechanisms that secure execution of untrusted code relies on is *interface-confinement*: untrusted code is often run inside a *sandbox* that confines its interaction with key system resources to a restricted set of interfaces. This practice is seen in web browsers, hypervisors, and other security-critical systems. For instance, ADsafe confines third-party scripts to limit their access to page content [3], [4]. A hypervisor limits an untrusted guest OS's physical memory accesses using address translation. Similarly, the TPM provides a set of APIs to access its PCRs, so that PCRs can only be extended with new values, but not arbitrarily overwritten.

Motivated by these systems, we present a program logic, called System M, for modeling and proving safety properties of systems that securely execute adversary-supplied code via interface-confinement. System M uses computation types to specify effects of computations. To verify safety properties [2], which are, by definition, predicates on traces, effects in computation types are specified by predicates over the entire trace of the computation. We call these predicates assertions. Actually, each assertion in System M is a pair of predicates on traces. One predicate, the standard *partial correctness assertion*, holds if the computation completes. The other, called the *invariant assertion*, holds at all intermediate points of the computation, even if the computation is stuck or divergent.

Inspired by HTT [5], we include a monad for suspended computations. The monadic type for suspended computations is of the form $\texttt{comp}(x{:}\tau.\varphi_1, \varphi_2)$, where φ_1 is the partial correctness assertion, τ is the type of the expression returned by the computation, and φ_2 is the invariant assertion.

Building on this basic infrastructure, we add a novel invariant type of form $\texttt{inv}(\varphi)$, to specify the properties of interface-confined code. The semantics of this type includes only terms that preserve the invariant φ on the trace, if they execute in an environment that provides arguments preserving the same invariant. This semantic definition captures the intuition behind interface-confinement: the effects of interface-confined code can be inferred by examining the effects of the interfaces. Adversary-supplied code that can only use a set of interfaces to generate effects satisfies the same invariant as the interfaces (see Section III-A for more details).

The novel typing rule Inv internalizes the intuition behind the analysis of interface-confined code and assigns an invariant type to terms that cannot perform effectful computations except through interfaces. This rule derives properties of untyped code provided by an adversary and, hence, enables the typing derivation of the trusted code to include assertions about effects of the adversarial code. The Inv rule generalizes prior work by Garg et al. on reasoning about interface-confined

DOI 10.1109/CSF.2015.38

adversarial code in a first-order language [6]. The main difference is that in this paper trusted interfaces can receive code (not just data) from the adversary and other trusted components, and it can execute the received code. Other frameworks like Bhargavan *et al.*'s contextual theorems [7] for F7 achieve expressiveness similar to the INV rule for a slightly limited selection of trace properties. (We compare to related work in Section VIII.)

System M is the first program logic that allows proofs of safety for programs that *execute* adversary-supplied code with adequate precautions, without requiring deep type (static) analysis of the adversarial code. In particular, we do not require a proof of safety to be shipped with adversarial code, as for example in proof-carrying code [8].

We construct a step-indexed model [9] over traces, and prove System M sound relative to the model. System M supports *compositional proofs*—security proofs of sequentially composed programs are built from proofs of their subprograms. System M also admits concurrent composition—properties proved of a program hold when that program executes concurrently with other, even adversarial, programs. Proving the soundness of System M's reasoning principles is challenging as System M combines dependent types with first-order reasoning about effects. Our step-indexed model for the invariant type and monadic type is novel. Our use of call-by-name β-reduction and the inclusion of untyped adversary-supplied code make the model nonstandard.

System M can be used to model and verify protocols as well as system designs. We demonstrate the reasoning principles of System M by manually constructing proofs of the state integrity property—that an attacker cannot roll back the state of trusted services—in the design of Memoir [10], a previously proposed trusted computing platform. Our case study reveals subtle assumptions that Memoir's security properties depend on, but are not explicitly mentioned in [10].

All technical details omitted from this paper can be found in our companion technical report [11].

II. TERM LANGUAGE AND OPERATIONAL SEMANTICS

A. Syntax

System M's term syntax is shown below.

Base values	bv	::=	$\mathtt{tt} \mid \mathtt{ff} \mid \iota \mid \ell \mid n \mid ()$
Expressions	e	::=	$x \mid bv \mid \lambda x.e \mid \mathtt{fix}\ f(x).e$
		$\mid$	$\Lambda X.e \mid e_1\ e_2 \mid e\ \cdot \mid \mathtt{comp}(c)$
Actions	a	::=	$A \mid a\ e \mid a\ \cdot$
Computations	c	::=	$\mathtt{act}(a) \mid \mathtt{ret}(e)$
		$\mid$	$\mathtt{letc}(c_1, x.c_2) \mid \mathtt{lete}(e_1, x.c_2)$
		$\mid$	$\mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2$

Pure expressions, denoted e, are distinguished from effectful computations, denoted c. An expression can be a variable, a base value, a function, a recursive function, a polymorphic function, a function application, a polymorphic function instantiation, or a suspended computation. We use $\cdot$ as the place holder for the type in a polymorphic function instantiation. Constants can be Booleans ($\mathtt{tt}, \mathtt{ff}$), natural numbers ($n \in \mathcal{N}$), thread identifiers ($\iota \in \mathcal{I}$), memory locations ($\ell \in \mathcal{L}$), and unit. System M is parameterized over a set of action symbols, denoted A, which are instantiated with concrete actions based for the specific application. For instance, A may be instantiated with memory operations such as read and write.

A basic computation is either an atomic action ($\mathtt{act}(a)$) or $\mathtt{ret}(e)$ that returns the pure expression e immediately. $\mathtt{letc}(c_1, x.c_2)$ denotes the sequential composition of c_1 and c_2, while $\mathtt{lete}(e_1, x.c_2)$ is the sequential composition of the suspended computation to which e_1 reduces and c_2. In both cases, the expression returned by the first computation is bound to x, which may occur free in c_2. We sometimes use the alternate syntax $\mathtt{letc}\ x = c_1; c_2$ and $\mathtt{let}\ x = e_1; c_2$. When the expression returned by the first computation is not used in c_2, we write $c_1; c_2$ and $e_1; c_2$. We also abbreviate $\mathtt{act}(a)$ to a.

B. Operational Semantics

We use System M to model systems that consist of several components executing concurrently. Therefore, the operational semantics of System M are small-step and allow interleaving of concurrent threads. The following syntactic constructs are used to to define configurations of concurrent systems.

Stack	K	::=	$[] \mid x.c :: K$
Thread	T	::=	$\langle \iota; K; c\rangle \mid \langle \iota; K; e\rangle \mid \langle \iota; \mathsf{stuck}\rangle$
Configuration	C	::=	$\sigma \triangleright T_1, \ldots, T_n$

A thread T is a unit of sequential execution. A non-stuck thread is a triple $\langle \iota; K; c\rangle$ or $\langle \iota; K; e\rangle$, where ι is a unique identifier of that thread (drawn from a set $\mathcal{I}$ of such identifiers), K is the execution (continuation) stack, and c (or e) is the computation (or expression) being evaluated currently. A thread permanently enters a stuck state, denoted $\langle \iota; \mathsf{stuck}\rangle$, after performing an illegal action, such as accessing an unallocated memory location. An execution stack is a list of frames of the form $x.c$ recording the return points of sequencing statements in the enclosing context. In a frame $x.c$, x binds the return expression of the computation preceding c. A configuration of the system, denoted C, consists of a shared state σ and a set of all threads. σ is application-specific; we assume that it contains at least a standard heap mapping memory locations to expressions, but it may contain more. For example, when modeling network protocols, σ may also contain the set of undelivered (pending) messages on the network.

The small-step transitions for threads and system configurations are shown in Figure 1. The relation $\sigma \triangleright T \hookrightarrow \sigma' \triangleright T'$ defines the small-step transition of a single thread. $\mathcal{C} \rightarrow \mathcal{C}'$ denotes the small-step transition for configuration $\mathcal{C}$; it results from the reduction of any single thread in $\mathcal{C}$.

The rules for $\sigma \triangleright T \hookrightarrow \sigma' \triangleright T'$ are mostly straightforward. The rules for evaluating an atomic action (R-ACTS and R-ACTF) rely on a function next that takes the current store σ and an action a, and returns a new store and an expression, which are the result of the action. If the action is illegal, then $\mathsf{next}(\sigma, a) = (\sigma', \mathsf{stuck})$. If the action returns a non-stuck expression e (rule R-ACTS), then the top frame $(x.c)$ is popped off the stack, and $c[e/x]$ becomes the current

$\boxed{\sigma \triangleright T \hookrightarrow \sigma' \triangleright T'}$

$$\frac{\mathsf{next}(\sigma, a) = (\sigma', e) \qquad e \neq \mathsf{stuck}}{\sigma \triangleright \langle \iota; x.c :: K; \mathtt{act}(a)\rangle \hookrightarrow \sigma' \triangleright \langle \iota; K; c[e/x]\rangle}\ \text{R-ACTS}$$

$$\frac{\mathsf{next}(\sigma, a) = (\sigma', \mathsf{stuck})}{\sigma \triangleright \langle \iota; x.c :: K; \mathtt{act}(a)\rangle \hookrightarrow \sigma' \triangleright \langle \iota; \mathsf{stuck}\rangle}\ \text{R-ACTF}$$

$$\frac{}{\sigma \triangleright \langle \iota; \mathsf{stuck}\rangle \hookrightarrow \sigma \triangleright \langle \iota; \mathsf{stuck}\rangle}\ \text{R-STUCK}$$

$$\frac{}{\sigma \triangleright \langle \iota; x.c :: K; \mathtt{ret}(e)\rangle \hookrightarrow \sigma \triangleright \langle \iota; K; c[e/x]\rangle}\ \text{R-RET}$$

$$\frac{}{\sigma \triangleright \langle \iota; K; \mathtt{lete}(e_1, x.c_2)\rangle \hookrightarrow \sigma \triangleright \langle \iota; x.c_2 :: K; e_1\rangle}\ \text{R-SEQE1}$$

$$\frac{e \rightarrow_\beta e'}{\sigma \triangleright \langle \iota; K; e\rangle \hookrightarrow \sigma \triangleright \langle \iota; K; e'\rangle}\ \text{R-SEQE2}$$

$$\frac{}{\sigma \triangleright \langle \iota; K; \mathtt{comp}(c_1)\rangle \hookrightarrow \sigma \triangleright \langle \iota; K; c_1\rangle}\ \text{R-SEQE3}$$

Fig. 1. Selected small-step reduction semantics of configurations

computation of the thread. If next returns stuck (rule R-ACTF), then the thread enters the stuck state and permanently remains there. When a sequencing statement $\mathtt{lete}(e_1, x.c_2)$ is evaluated, the frame $x.c_2$ is pushed onto the stack, and e_1 is first reduced to a suspended computation $\mathtt{comp}(c_1)$; then c_1 is evaluated. Pure expressions are reduced using standard call-by-name β-reduction rules ($\rightarrow_\beta$). This choice is explained in Sections V-D.

Any *finite* execution from a configuration results in a trace $\mathcal{T}$, defined as a finite sequence of reductions. With each reduction we associate a *time point* u. The time point represents the clock time (a natural number) at which the reduction happens. Time points on a trace are monotonically increasing. A trace annotated with time is written $\xrightarrow{u_0} C_0 \xrightarrow{u_1} C_1 \ldots \xrightarrow{u_n} C_n$, where $u_i < u_{i+1}$. We follow the convention that the reduction from C_i to C_{i+1} happens at time u_{i+1} and that its effects occur immediately. Thus the state at time u_i is the state in C_i. Note that our operational semantics do not stipulate the time gaps between two consecutive reductions and, hence, do not determine the time points associated with a trace. Instead, a trace (without time points) generated by the operational semantics corresponds to all its annotations with time points that satisfy monotonicity. In our assertions and proofs, we use time points only to specify a relative order between events on a trace, so their concrete values are irrelevant. Also, time points are distinct from the number of steps on a trace; we use the latter as the basis for step-indexing and as an induction measure in our soundness proofs.

III. Overview of Interface Confinement

In analyzing security properties of programs that execute adversary-supplied code, one often encounters a partially trusted program, whose code is *unknown*, but which is *known* or assumed to be confined to the use of a specific set of interfaces to perform actions on shared state. Using just the knowledge of confinement, we can sometimes deduce the effects of the partially trusted program. We now introduce an example to illustrate interface confinement and its semantics.

$$\begin{array}{ll} inc & = \mathtt{comp}(\mathtt{letc}\ x = \mathsf{read}\ cnt; \mathsf{write}\ cnt\ (x{+}1)) \\ get & = \mathtt{comp}(\mathtt{letc}\ x = \mathsf{read}\ cnt; \mathtt{ret}(x)) \\ prn & = \lambda y.\mathtt{comp}(\mathtt{lete}\ \_ = y \\ & \qquad \mathtt{letc}\ x = \mathsf{read}\ \ cnt; \mathsf{print}\ \ x) \\ c & = \mathtt{letc}\ x = \mathsf{download}(); \mathtt{letc}\ y = \mathsf{check}\ x; \\ & \quad \mathtt{lete}\ \_ = y\ inc\ get\ prn; \mathtt{ret}() \end{array}$$

Fig. 2. Example counter program

A. Interfaces for a Counter

Consider the following example. A memory location cnt stores a counter value, which is intended to be non-decreasing. To ensure that the counter never decreases, we confine the untrusted client to only the following interfaces for access to cnt: the inc interface that increments the counter value by 1; the get interface that returns the counter value, and the prn interface that takes a computation as its argument, executes the computation and then prints out the counter value. The interfaces are shown in Figure 2.

Now, consider a use of this counter. Some thread ι runs the computation c, whose code is also shown in Figure 2. The computation c first downloads an untrusted expression which it binds to x and checks that the downloaded expression contains no actions (action check x). Next, the untrusted expression is applied to the three interfaces for access to cnt. The untrusted expression may combine the three given interfaces in any way it likes. Irrespective of what the untrusted expression does, its execution preserves the *invariant* that the counter value never decreases because none of the three interfaces decrement the counter and the untrusted expression has no actions in it syntactically (so it cannot read or write cnt directly).

The computation c is a specific instance of a more general scenario where a trusted component downloads code from an untrusted, possibly adversarial source and confines the execution of the untrusted code to a set of interfaces. The untrusted code is checked lightly to ensure that it has no instructions to cause side-effects, but it may combine the provided interfaces in arbitrary ways. The guarantee we have is that the untrusted code's execution cannot violate any invariant that is common to all the provided interfaces.

B. Reasoning about Effects from Confinement

In System M, we use the type $\mathtt{inv}(\varphi)$ to classify expressions whose evaluation (including the evaluation of any nested computations c) *preserves* the invariant φ. We call this type an *invariant type*. The denotation of this type, $\mathcal{RE}[\![\mathtt{inv}(\varphi)]\!]$, is the set of expressions that preserve the invariant φ. The invariant type is intended to classify interface-confined *adversary-supplied code*, which could include ill-typed terms such as an application of an integer to a function, so expressions of

invariant types may be, or may reduce to, stuck terms that are not values. This makes the type $\texttt{inv}(\varphi)$ very distinct from other types in System M, and also from types in standard type systems (which usually ensure non-stuckedness).

Although formal definitions are introduced in Section IV-B, we observe a few salient points here. First, if a function lies in $\mathcal{RE}[\![\texttt{inv}(\varphi)]\!]$, then any suspended computation it returns must also preserve the invariant φ. If not, then the adversary can break the invariant by executing the returned computation. Second, when we check that a function preserves an invariant, we assume that any expressions in its arguments also preserve invariant, else the function (which is possibly adversarial) must be assumed to not execute the given expressions. This means that if a function has type $\texttt{inv}(\varphi)$, then its arguments and its result have the same type. This introduces a circularity in the definition of $\mathcal{RE}[\![\texttt{inv}(\varphi)]\!]$, which we break using step-indexing. Technically, $\mathcal{RE}[\![\texttt{inv}(\varphi)]\!]$ is a set of pairs of the form (k, e) where e is an expression and k is a step-index. The intuition here is that if $(k, e) \in \mathcal{RE}[\![\texttt{inv}(\varphi)]\!]$, then e's execution preserves the invariant φ and, additionally, if e returns in less than k steps, then the result also preserves the same invariant. The informal definition of $\mathcal{RE}[\![\texttt{inv}(\varphi)]\!]$ is shown below.

$$
\begin{aligned}
&\mathcal{RE}[\![\texttt{inv}(\varphi)]\!] = \\
&\quad \{(k, \texttt{comp}(c)) \mid \text{effects of } c \text{ satisfy } \varphi, \\
&\qquad\qquad \text{and if } c \text{ returns } e \text{ in } j \text{ steps}, \\
&\qquad\qquad (k{-}j, e) \in \mathcal{RE}[\![\texttt{inv}(\varphi)]\!]\} \\
&\quad \cup \{(k, \lambda x.e) \mid \forall e', j, j{<}k, (j, e'){\in}\mathcal{RE}[\![\texttt{inv}(\varphi)]\!] \\
&\qquad\qquad \Longrightarrow (j, e[e'/x]) \in \mathcal{RE}[\![\texttt{inv}(\varphi)]\!]\} \\
&\quad \cup \{(k, \texttt{nf}) \mid \texttt{nf} \text{ is a stuck term that is not in} \\
&\qquad\qquad \text{introduction form}\} \cdots
\end{aligned}
$$

Observe that in the λ case, where we recursively invoke the definition of $\mathcal{RE}[\![\texttt{inv}(\varphi)]\!]$, the step-index j is strictly less than the original step index k. This breaks the circularity. Also, $\mathcal{RE}[\![\texttt{inv}(\varphi)]\!]$ includes all normal terms, including stuck terms, which would not be typeable in a conventional type system.

IV. Types and Their Semantics

A. Types

Figure 3 summaries System M's types. Types for expressions are denoted by τ and types for computations are denoted by η. The computation type η is a pair consisting of a partial correctness assertion and an invariant assertion. Assertions, denoted φ, are standard first-order logical formulas interpreted over traces. The partial correctness assertion $(x{:}\tau.\varphi)$ describes the effect of a computation if it terminates: the computation will produce effects that satisfy φ and return an expression of type τ. The invariant assertion describes the effects of a computation while it is still being evaluated (not returned yet).

The computation type η is parameterized by the time interval $(u_b, u_e]$ over which the computation has executed. The assertions in η use self to refer to the ID of the thread that executes the computation. (The expression self only appears in assertions). A closed computation type $(\Xi.x{:}\tau.\varphi_1, \Xi.\varphi_2)$ binds u_b and u_e in Ξ. For brevity, we omit the types in Ξ and simply write (u_b, u_e). Expression types include type variables (X), base types $\texttt{b}$, dependent function types $(\Pi x{:}\tau_1.\tau_2)$, and polymorphic function types $(\forall X.\tau)$. The type $\texttt{any}$ contains all syntactically well-formed expressions ($\texttt{any}$ stands for "untyped"). Adversary-supplied code, such as an expression read from memory, is initially typed $\texttt{any}$. A suspended computation $\texttt{comp}(c)$ is assigned a monadic type $\texttt{comp}(\eta_c)$.

Base types	$\texttt{b}$	$::= \texttt{bool} \mid \texttt{nat} \mid \texttt{unit} \mid \texttt{ptr} \mid \texttt{time} \mid \texttt{thread}$
Expr types	τ	$::= X \mid \texttt{b} \mid \Pi x{:}\tau_1.\tau_2 \mid \forall X.\tau \mid \texttt{comp}(\eta_c)$
		$\mid \texttt{any} \mid \texttt{inv}(\Xi.\varphi) \mid \texttt{FAE}$
Comp types	η	$::= (x{:}\tau.\varphi, \varphi')$
Closed c types	η_c	$::= (\Xi.x{:}\tau.\varphi_1, \Xi.\varphi_2)$
Assertions	φ	$::= P \mid e_1 = e_2 \mid \varphi\ e \mid \top \mid \bot \mid \neg\varphi$
		$\mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \forall x{:}\tau.\varphi \mid \exists x{:}\tau.\varphi$
Expressions	e	$::= \cdots \mid \mathsf{self}$
Exec ctx	Ξ	$::= (u_b : \texttt{time}, u_e : \texttt{time})$

Fig. 3. Types in System M

The invariant type that we discussed in Section III-A is denoted $\texttt{inv}(\Xi.\varphi)$. The execution context Ξ binds the time interval during which the computation executes. We use the type $\texttt{FAE}$ for expressions that syntactically do not include any action symbols. The adversary-supplied code in our example (Section III-A) is typed $\texttt{FAE}$ after the dynamic check.

B. Semantics

We define step-indexed semantics for types and assertions. The step-indices are necessary for defining the semantics of the invariant type $\texttt{inv}(\Xi.\varphi)$ and for proving the soundness of recursive function typing. The interpretation of an expression type τ is a semantic type, written C. Each C is a set of pairs; each pair contains a step-index and an expression. The expression has to be in normal form, denoted $\texttt{nf}$. Expressions in normal form cannot be reduced further under call-by-name β-reduction. We require that C be closed under reduction of step-indices. The set of all semantic types is denoted Type.

$$
\begin{aligned}
\mathsf{Type} \stackrel{def}{=} \{\mathsf{C} \mid\ & \mathsf{C} \in \mathcal{P}(\{(j, \texttt{nf}) \mid j \in \mathbb{N}\}) \wedge \\
& (\forall k, \texttt{nf}, (k, \texttt{nf}) \in \mathsf{C} \wedge j < k \Longrightarrow (j, \texttt{nf}) \in \mathsf{C})\}
\end{aligned}
$$

We define the *value and expression interpretations* of expression types τ (written $\mathcal{RV}[\![\tau]\!]$ and $\mathcal{RE}[\![\tau]\!]$), as well as the *interpretation* of computation types η (written $\mathcal{RC}[\![\eta]\!]$) simultaneously. Formulas are interpreted on traces. We write $\mathcal{T} \vDash \varphi$ to mean that φ is true on trace $\mathcal{T}$.

Interpretation of computation types The interpretation of a computation type, $\mathcal{RC}[\![\eta]\!]^{\mathcal{K},\iota}_{\theta;\mathcal{T}}$, is a set of step-indexed computations (k, c). The substitution θ denotes a partial map from type variables to Type. Other parameters in the $\mathcal{RC}[\![\eta]\!]$ relation are illustrated in Figure 4. The trace $\mathcal{T}$ contains c's execution. $\mathcal{K}{=}(t_b, t_e)$ identifies the time interval during which c executes. ι is the identifier of the thread that executes c. The semantics of computation types are the intersection of two sets—the first set contains indexed computations that satisfy the partial correctness assertion $x{:}\tau.\varphi_1$, and the second set contains computations that satisfy the invariant assertion φ_2.

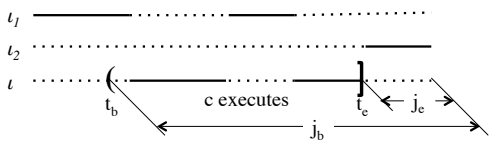

Fig. 4. The above trace consists of three threads. Solid lines illustrate steps when a thread is executing. Dashed lines illustrate idle steps of a thread.

$$\begin{aligned}
&\mathcal{RC}[\![(x{:}\tau.\varphi_1, \varphi_2)]\!]^{\mathcal{K},\iota}_{\theta;\mathcal{T}} = \\
&\quad \{(k,c) \mid (\iota, c, \mathcal{T}\downarrow_{\mathcal{K}})\Downarrow(e', j_b, j_e) \text{ and } k \geq j_b \geq j_e \\
&\qquad \Longrightarrow (j_e, e') \in \mathcal{RE}[\![\tau]\!]^{E(\mathcal{K})}_{\theta;\mathcal{T}} \text{ and } \mathcal{T} \vDash \varphi_1[e'/x]\} \\
&\quad \cap\{(k,c) \mid (\iota, c, \mathcal{T}\downarrow_{\mathcal{K}})\Uparrow(j_b, j_e) \text{ and } k \geq j_b \geq j_e \\
&\qquad \Longrightarrow \mathcal{T} \vDash \varphi_2\}
\end{aligned}$$

We write $(\iota, c, \mathcal{T}\downarrow_{\mathcal{K}})\Downarrow(e', j_b, j_e)$ to denote that thread ι executes c in the time interval $(t_b, t_e]$ on the trace $\mathcal{T}$, where t_b is the first time when c becomes available to be executed by ι, c returns e' at time t_e, and j_b and j_e are the lengths of $\mathcal{T}$ from time t_b and t_e to the end of $\mathcal{T}$, respectively. We write $(\iota, c, \mathcal{T}\downarrow_{\mathcal{K}})\Uparrow(j_b, j_e)$ to denote that thread ι executes c in the time interval $(t_b, t_e]$ on $\mathcal{T}$, where t_b is the first time when c becomes available to be executed by ι, and c has not returned at time t_e (j_b and j_e have the same meaning as before).

The partial correctness assertion requires that $\mathcal{T}$ satisfy $\varphi_1[e'/x]$ and that e' be of type τ semantically. Here the index k has to be large enough so that the last k steps of $\mathcal{T}$ contains the complete execution of c. The index of the return expression is j_e, since e' will be evaluated in the remaining steps. Here, $E(\mathcal{K})$ denotes t_e when $\mathcal{K} = (t_b, t_e)$. (We explain the meaning of the time point in $\mathcal{RE}[\![\tau]\!]$ in the next paragraph.) The invariant assertion requires that $\mathcal{T}$ satisfy φ_2.

Interpretation of expression types The value and expression interpretations of types are defined in Figure 5. θ and $\mathcal{T}$ have the same meanings as before. t is the time after which e reduces. The interpretation of the function type $\Pi x{:}\tau_1.\tau_2$ is nonstandard: the substitution for the variable x is an expression, not a value. Since System M uses call-by-name β-reduction, the reduction of $e_1\, e_2$ need not evaluate e_2 before it is supplied to the function that e_1 reduces to. Further, the definition builds-in both step-index downward closure and time delay: given any argument e' that has a smaller index j and evaluates after t', which is later than t, the function application belongs to the interpretation of the argument type with the index j and time point t'.

The interpretation of the monadic type includes suspended computations $(k, \mathtt{comp}(c))$ such that (k, c) belongs to the interpretation of computation types, defined earlier. Because c executes after time t, the beginning and ending time points selected for evaluating c are no earlier than t. We write $t \leq \mathcal{K}$ to denote $t \leq t_b \leq t_e$, given $\mathcal{K} = (t_b, t_e)$. The interpretation of the any type contains all normal forms.

The semantics for the invariant type $\mathtt{inv}(\Xi.\varphi)$ is the union of five sets. The first set includes all stuck terms that are not in introduction form. These terms can neither be evaluated further, nor can they be eliminated. As a result, there is no need to check the assertion φ against the trace. For instance, $(k, 5\ \mathtt{tt}) \in \mathcal{RV}[\![\mathtt{inv}(\Xi.\bot)]\!]^t_{\theta;\mathcal{T}}$, even though $\bot$ does not hold on any trace. The next four sets include indexed functions and suspended computations. The definitions require that these expressions preserve an invariant if they are evaluated in a context that preserves the same invariant.

$$\begin{aligned}
&\mathcal{RV}[\![\mathtt{any}]\!]^t_{\theta;\mathcal{T}} = \{(k, \mathtt{nf}) \mid k \in \mathbb{N}\} \\
&\mathcal{RV}[\![X]\!]^t_{\theta;\mathcal{T}} = \theta(X) \\
&\mathcal{RV}[\![\Pi x{:}\tau_1.\tau_2]\!]^t_{\theta;\mathcal{T}} = \\
&\quad \{(k, \lambda x.e) \mid \forall j < k, \forall t' \geq t, \forall e', (j,e') \in \mathcal{RE}[\![\tau_1]\!]^{t'}_{\theta;\mathcal{T}} \\
&\qquad \Longrightarrow (j, e[e'/x]) \in \mathcal{RE}[\![\tau_2[e'/x]]\!]^{t'}_{\theta;\mathcal{T}}\} \cup \\
&\quad \{(k, \mathtt{fix}\ f(x).e) \mid \forall j < k, \forall t' \geq t, \forall e', (j,e') \in \mathcal{RE}[\![\tau_1]\!]^{t'}_{\theta;\mathcal{T}} \\
&\qquad \Longrightarrow (j, e[e'/x][\mathtt{fix}\ f(x).e/f]) \in \mathcal{RE}[\![\tau_2[e'/x]]\!]^{t'}_{\theta;\mathcal{T}}\} \\
&\mathcal{RV}[\![\forall X.\tau]\!]^t_{\theta;\mathcal{T}} = \\
&\quad \{(k, \Lambda X.e) \mid \forall j{<}k, \forall \mathsf{C} \in \mathsf{Type} \Longrightarrow (j,e) \in \mathcal{RE}[\![\tau]\!]^t_{\theta[X \mapsto \mathsf{C}];\mathcal{T}}\} \\
&\mathcal{RV}[\![\mathtt{comp}(\Xi.x{:}\tau.\varphi_1, \Xi.\varphi_2)]\!]^t_{\theta;\mathcal{T}} = \\
&\quad \{(k, \mathtt{comp}(c)) \mid \forall \mathcal{K}, \iota, t \leq \mathcal{K}, \text{let } \gamma = [\mathcal{K}, \iota/\Xi, \mathsf{self}] \\
&\qquad (k,c) \in \mathcal{RC}[\![((x{:}\tau.\varphi_1)\gamma, \varphi_2\gamma)]\!]^{\mathcal{K},\iota}_{\theta;\mathcal{T}}\} \\
&\mathcal{RV}[\![\mathtt{inv}(\Xi.\varphi)]\!]^t_{\theta;\mathcal{T}} = \\
&\quad \{(k, \mathtt{nf}) \mid \mathtt{nf} \neq \lambda x.e, \mathtt{fix}\ \ f(x) = e, \Lambda X.e, \mathtt{comp}(c)\} \\
&\quad \cup\{(k, \mathtt{comp}(c)) \mid \forall \mathcal{K}, \iota, t \leq \mathcal{K}, \text{let } \gamma = [\mathcal{K}, \iota/\Xi, \mathsf{self}] \\
&\qquad (k,c) \in \mathcal{RC}[\![((x{:}\mathtt{inv}(\Xi.\varphi).\varphi)\gamma, \varphi\gamma)]\!]^{\mathcal{K},\iota}_{\theta;\mathcal{T}}\} \\
&\quad \cup\{(k, \lambda x.e) \mid \forall j{<}k, \forall t'{\geq}t, \forall e', (j,e') \in \mathcal{RE}[\![\mathtt{inv}(\Xi.\varphi)]\!]^{t'}_{\theta;\mathcal{T}} \\
&\qquad \Longrightarrow (j, e[e'/x]) \in \mathcal{RE}[\![\mathtt{inv}(\Xi.\varphi)]\!]^{t'}_{\mathcal{T}}\} \\
&\quad \cup\{(k, \mathtt{fix}\ f(x).e) \mid \\
&\qquad \forall j{<}k, \forall t' \geq t, \forall e', (j,e') \in \mathcal{RE}[\![\mathtt{inv}(\Xi.\varphi)]\!]^{t'}_{\theta;\mathcal{T}} \\
&\qquad \Longrightarrow (j, e_1[e'/x][\mathtt{fix}\ f(x).e/f]) \in \mathcal{RE}[\![\mathtt{inv}(\Xi.\varphi)]\!]^{t'}_{\theta;\mathcal{T}}\} \\
&\quad \cup\{(k, \Lambda X.e) \mid \forall j < k \Longrightarrow (j,e) \in \mathcal{RE}[\![\mathtt{inv}(\Xi.\varphi)]\!]^t_{\theta;\mathcal{T}}\} \\
&\mathcal{RV}[\![\mathtt{FAE}]\!]^t_{\theta;\mathcal{T}} = \{(k, \mathtt{nf}) \mid k \in \mathbb{N} \text{ and } \emptyset \vdash \mathtt{nf} : \mathtt{FAE}\}
\end{aligned}$$

$$\begin{aligned}
&\mathcal{RE}[\![\tau]\!]^t_{\theta;\mathcal{T}} = \{(k,e) \mid \forall j \leq k, \forall e', e \to^j_\beta e' \nrightarrow_\beta \\
&\qquad \Longrightarrow (k-j, e') \in \mathcal{RV}[\![\tau]\!]^t_{\theta;\mathcal{T}}\}
\end{aligned}$$

Fig. 5. Semantics for expression types

The interpretation of FAE includes all normal forms that syntactically do not contain any actions. We define rules $\Gamma \vdash e : \mathtt{FAE}$ to check that all the free variables in e are in Γ and that e does not contain action symbols in its syntax.

We lift the value interpretation $\mathcal{RV}[\![\tau]\!]^t_{\theta;\mathcal{T}}$ to the expression interpretation $\mathcal{RE}[\![\tau]\!]^t_{\theta;\mathcal{T}}$ in a standard way (bottom of Figure 5).

All our interpretations are well-defined by induction on step-indices. Note that the definition of $\mathcal{RV}[\![\mathtt{inv}(\Xi.\varphi)]\!]^t_{\theta;\mathcal{T}}$ appeals to the definition of $\mathcal{RC}[\![((x{:}\mathtt{inv}(\Xi.\varphi).\varphi)\gamma, \varphi\gamma)]\!]^{\mathcal{K},\iota}_{\theta;\mathcal{T}}$ without decrementing the step-index, but this does not lead to a circularity because the latter invokes the definition of $\mathcal{RE}[\![\mathtt{inv}(\Xi.\varphi)]\!]$ only at a strictly smaller step-index (a computation takes at least one step to return an expression).

Formula semantics Formulas are interpreted over traces. The trace semantics of formulas are mostly standard. We show a few key definitions below. We assume a valuation function $\varepsilon(\mathcal{T})$ that returns the set of atomic formulas that are true on the trace $\mathcal{T}$. We define the predicate $\mathtt{start}(e_1, \mathtt{comp}(c), e_2)$ to mean that the starting time of the execution of the computation c, by the thread with ID that e_1 reduces to, is e_2. This predicate

is used by the type system (Section VI).

$\mathcal{T} \vDash P\ \vec{e}$ iff $P\ \vec{e} \in \varepsilon(\mathcal{T})$
$\mathcal{T} \vDash \mathsf{start}(e_1, \mathtt{comp}(c), e_2)$ iff $e_1 \to_\beta^* \iota \not\to_\beta, e_2 \to_\beta^* t \not\to_\beta$, and thread ι has c as the active computation with an empty stack at time t on $\mathcal{T}$
$\mathcal{T} \vDash \forall x{:}\tau.\varphi$ iff $\forall e, (\_, e) \in [\![\tau]\!]$ implies $\mathcal{T} \vDash \varphi[e/x]$
$\mathcal{T} \vDash \exists x{:}\tau.\varphi$ iff $\exists e, (\_, e) \in [\![\tau]\!]$ and $\mathcal{T} \vDash \varphi[e/x]$

The universal and existential quantifiers quantify over terms of base types. The index and θ, $\mathcal{T}$, and t do not matter for the interpretation of base types, so we write $[\![\tau]\!]$ as short hand for $\exists \theta, \mathcal{T}, t, \mathcal{RE}[\![\tau]\!]^t_{\theta;\mathcal{T}}$.

C. Examples

We continue our example from Section III-A. Let the predicate $\mathsf{mem}\ l\ v\ t$ mean that at time t, memory location l stores value v and let $\mathsf{eval}\ e\ e'$ be true if e β-reduces to e' and e' is in normal form. The following invariant says that the counter cnt does not decrease in the time interval $[u_b, u_e]$.

$$\varphi_{nd}(u_b, u_e) = \forall t_1, t_2, l, v_1, v_2, u_b{\le}t_1{<}t_2{\le}u_e \wedge \mathsf{eval}\ cnt\ l \wedge \mathsf{mem}\ l\ v_1\ t_1 \wedge \mathsf{mem}\ l\ v_2\ t_2 \Rightarrow v_2 \ge v_1$$

Consider some code e provided by the adversary. We would like to show that e, when executed with the interfaces inc and prn in some thread ι preserves this invariant. However, this depends on other threads in the configuration as well. We make the assumption that no other thread modifies cnt. This is formalized by Δ_1 below.

$$\Delta_1{=}\forall i, e, l, v, t, \mathsf{eval}\ cnt\ l \wedge \mathsf{write}\ i\ e\ v\ t \wedge \mathsf{eval}\ e\ l \Rightarrow \mathsf{eval}\ i\ \iota$$

We define the invariant $\varphi_{nd1}(u_b, u_e)$ to include the assumption that ι is the ID of the thread that executes e.

$$\varphi_{nd1}(u_b, u_e) = \mathsf{self}{=}\iota \Rightarrow \varphi_{nd}(u_b, u_e)$$

Expression e preserves the above invariant:

$$(k, e\ inc\ get\ prn) \in \mathcal{RE}[\![\mathtt{inv}((u_b, u_e).\varphi_{nd})]\!]^t_{\theta;\mathcal{T}} \qquad (1)$$

To prove it, we need the assumption that e does not contain any actions (to ensure that e accesses cnt only through the three provided interfaces). This is ensured by checking that e has the type FAE using the type system of Section V.

We can prove the following lemma.

Lemma 1. *If* $\emptyset \vdash e : \mathtt{FAE}$ *then* $\forall k, t, \mathcal{T}$*, s.t.,* $\mathcal{T} \vDash \Delta_1$*,* $(k, e) \in \mathcal{RE}[\![\mathtt{inv}(\Xi.\varphi_{nd})]\!]^t_{\theta;\mathcal{T}}$.

Using the definition of $\mathcal{RE}[\![\mathtt{inv}(\Xi.\varphi)]\!]$, we can prove as a corollary that (1) holds.

V. Type System and Assertion Logic

Typing contexts and judgments Our typing judgments use several contexts. Θ contains type variables. The signature Σ contains specifications for action symbols. We write α to denote the type for partially and fully-applied actions. As these actions can be invoked by any thread at any time, we use the closed computation type to classify fully-applied actions ($\mathtt{Act}(\eta_c)$). Γ contains variable type bindings. Δ contains logical assertions. In computation typing, the ordered context $\Xi = (u_b, u_e)$ represent the interval $(u_b, u_e]$ during which the computation executes and self is a name for the ID of the thread that runs that computation.

For technical convenience, we define a relation $\Xi \to (\Xi_0, \Delta_0) \rhd (\Xi_1, \Delta_1)$ on the five contexts $\Xi, \Xi_0, \Delta_0, \Xi_1, \Delta_1$ by the following single rule:

$$(u_b, u_e) \to ((u_b, u_m), u_b{\le}u_m{\le}u_e) \rhd ((u_m, u_b), u_b{\le}u_m{\le}u_e)$$

Here u_m is a fresh parameter. Our typing rules use the above relation to split an execution contexts into several adjacent contexts, where the Δs record ordering constraints. We write $\Xi \to (\Xi_0, \Delta_0) \rhd (\Xi_1, \Delta_1) \rhd (\Xi_2, \Delta_2)$ as a short hand for $\Xi \to (\Xi_0, \Delta_0) \rhd (\Xi'_1, \Delta'_1)$, $\Xi'_1 \to (\Xi_1, \Delta''_1) \rhd (\Xi_2, \Delta'_2)$, $\Delta_1 = \Delta'_1, \Delta''_1$, and $\Delta_2 = \Delta'_1, \Delta'_2$.

A summary of the typing judgments is shown below.

$u; \Theta; \Sigma; \Gamma; \Delta \vdash e : \tau$	expression e has type τ
$\Xi; \Theta; \Sigma; \Gamma; \Delta \vdash c : \eta$	computation c has type η
$\Xi; \Theta; \Sigma; \Gamma; \Delta \vdash \varphi\ \mathsf{silent}$	φ holds while reductions are non-effectful
$\Theta; \Sigma; \Gamma; \Delta \vdash \varphi\ \mathsf{true}$	φ is true

Silent or *non-effectful* reductions are pure reductions, i.e., reductions of terms other than actions.

Predicates used in examples Each action has a corresponding *action predicate*, which specifies when that action happens, by which thread, and with what arguments. For instance, the action predicate that corresponds to the read action is $\mathsf{Read}\ i\ l\ v\ t$, denoting that thread i reads location l at time t, and the read action returns value v, (which must be the value stored at location l at time t). We follow the convention that the first argument of an action predicate is the ID of the thread that executes that action and the last argument is the time at which the action happens. Predicate $\mathsf{noActions}\ i\ t$ means that thread i is silent at time t (i.e., does not perform any actions at t). We use the following abbreviations to describe effects during a time interval.

$$P@(t_1, t_2) = \forall t, t_1{<}t{<}t_2, P\ t$$
$$P@(t_1, t_2] = \forall t, t_1{<}t{\le}t_2, P\ t$$

Logical reasoning System M includes a proof system for first-order logic, most of which is standard. The first-order logic enables reasoning about the properties of traces and is used in weakening postconditions of computations, as explained later. System M has two rules for reasoning about programs running concurrently; which are explained in Section VI.

Silent threads The typing judgment $\Xi; \Theta; \Sigma; \Gamma; \Delta \vdash \varphi\ \mathsf{silent}$ specifies properties of threads while they perform only non-effectful reductions or do not reduce at all. The judgment is auxiliary in proofs of both partial correctness and invariant assertions. The following rule states that if φ is true, then a trace containing a thread's silent computation satisfies φ.

$$\frac{\Theta; \Sigma; \Xi, \Gamma; \Delta \vdash \varphi\ \mathsf{true} \qquad \Xi, \Gamma \vdash \varphi\ \mathsf{ok}}{\Xi; \Theta; \Sigma; \Gamma; \Delta \vdash \varphi\ \mathsf{silent}}\ \textsc{Silent}$$

$$\frac{\Xi;\Theta;\Sigma;\Gamma;\Delta \vdash a :: \texttt{Act}(\Xi'.x{:}\tau.\varphi_1, \Xi'.\varphi_2) \quad \Xi;\Theta;\Sigma;\Gamma;\Delta \vdash \varphi \text{ silent} \quad \texttt{fv}(a) \in \texttt{dom}(\Gamma)}{\Xi;\Theta;\Sigma;\Gamma;\Delta \vdash \texttt{act}(a) : (x{:}\tau.\varphi_1[\Xi/\Xi'], \varphi_2[\Xi/\Xi'] \wedge \varphi)}\ \textsc{Act}$$

$$\frac{E(\Xi);\Theta;\Sigma;B(\Xi);\Gamma;\Delta \vdash e : \tau \quad \Xi;\Theta;\Sigma;\Gamma;\Delta \vdash \varphi \text{ silent} \quad \texttt{fv}(e) \subseteq \texttt{dom}(\Gamma)}{\Xi;\Theta;\Sigma;\Gamma;\Delta \vdash \texttt{ret}(e) : (x{:}\tau.((x = e) \wedge \varphi), \varphi)}\ \textsc{Ret}$$

$$\frac{\begin{array}{l}\Theta;\Sigma;\Xi,\Gamma;\Delta \vdash (y : \tau'.\varphi_P, \varphi_I) \text{ ok} \\ \texttt{fv}(\texttt{letc}(c_1, x.c_2)) \subseteq \texttt{dom}(\Gamma) \\ \Xi \to (\Xi_0, \Delta_0) \rhd (\Xi_1, \Delta_1) \rhd (\Xi_2, \Delta_2) \\ \Xi_0;\Theta;\Sigma;\Xi\backslash\Xi_0,\Gamma;\Delta,\Delta_0 \vdash \varphi_0 \text{ silent} \\ \Xi_1;\Theta;\Sigma;\Xi \cup \Xi_0\backslash\Xi_1,\Gamma;\Delta,\Delta_1,\varphi_0 \\ \quad \vdash c_1 : (x{:}\tau.\varphi_{P1}, \varphi_{I1}) \\ \Xi_2;\Theta;\Sigma;\Xi \cup \Xi_0 \cup \Xi_1\backslash\Xi_2,\Gamma,x:\tau;\Delta,\Delta_2,\varphi_0,\varphi_{P1} \\ \quad \vdash c_2 : (y : \tau'.\varphi_{P2}, \varphi_{I2}) \\ \Theta;\Sigma;\Xi,\Gamma;\Delta \vdash \varphi_0[\Xi/\Xi_0] \Rightarrow \varphi_I \text{ true} \\ \Theta;\Sigma;\Xi \cup \{B(\Xi_1)\},\Gamma;\Delta,\Delta_1[E(\Xi)/E(\Xi_1)] \\ \quad \vdash \varphi_0 \wedge \varphi_{I1}[E(\Xi)/E(\Xi_1)] \Rightarrow \varphi_I \text{ true} \\ \Theta;\Sigma;\Xi \cup \Xi_0 \cup \Xi_1 \cup \Xi_2,\Gamma,x:\tau;\Delta,\Delta_2 \\ \quad \vdash \varphi_0 \wedge \varphi_{P1} \wedge \varphi_{I2} \Rightarrow \varphi_I \text{ true} \\ \Theta;\Sigma;\Xi \cup \Xi_0 \cup \Xi_1 \cup \Xi_2,\Gamma,x:\tau;\Delta,\Delta_2 \\ \quad \vdash \varphi_0 \wedge \varphi_{P1} \wedge \varphi_{P2} \Rightarrow \varphi_P \text{ true}\end{array}}{\Xi;\Theta;\Sigma;\Gamma;\Delta \vdash \texttt{letc}(c_1, x.c_2) : (y : \tau'.\varphi_P, \varphi_I)}\ \textsc{SeqC}$$

Fig. 6. Selected Computation Typing Rules

The type system may be extended with other sound rules for this judgment. For instance, the following is a sound rule stating that when the a thread is silent during the interval $(u_b, u_e]$, then it performs no actions during that interval.

SILENT-NA

$$\frac{}{(u_b, u_e); \Sigma; \Theta; \Gamma; \Delta \vdash \mathsf{noActions}\ \mathsf{self}@(u_b, u_e] \text{ silent}}$$

A. Typing Computations

Figure 6 shows selected rules for establishing partial correctness postconditions and invariant properties of computations. The judgment $(u_b, u_e); \Theta; \Sigma; \Gamma; \Delta \vdash c : (x{:}\tau.\varphi_1, \varphi_2)$ means that if in a trace $\mathcal{T}$ a thread with id ι begins to execute computation c at time t_b, and at time t_e, c returns an expression e, then e has type τ, and $\mathcal{T}$ satisfies $\varphi[t_b, t_e, \iota, e/u_b, u_e, \mathsf{self}, x]$ (recall that self is the placeholder for the ID of the thread that runs c); and that if in a trace $\mathcal{T}$ a thread with id ι begins to execute computation c at time t_b, and at time t_e, c has not returned, then $\mathcal{T}$ satisfies $\varphi[t_b, t_e, \iota/u_b, u_e, \mathsf{self}]$.

The type of an atomic action is directly derived from the specification of the action symbol in a in rule ACT. We elide rules for the judgment $a :: \texttt{Act}(\Xi'.x{:}\tau.\varphi_1, \Xi'.\varphi_2)$, which derives types for actions based on the specifications in Σ. The atomic action executes within the interval specified by Ξ, so Ξ substitutes Ξ' in the resulting assertions. The thread executing the atomic action is silent before the action returns, and thus, the invariant assertion of the action is the conjunction of the invariant specified in Σ and the effect of being silent.

For our examples, Σ includes the following specifications:

$$\begin{array}{ll}\mathsf{read}: & \Pi x{:}\texttt{ptr}.\texttt{Act}((u_b,u_e).y{:}\texttt{any}.u_e{>}u_b \wedge \mathsf{Read\ self}\ x\ y\ u_e \\ & \qquad \wedge \mathsf{noActions\ self}@(u_b,u_e), \\ & \quad (u_b,u_e).\top) \\ \mathsf{write}: & \Pi x{:}\texttt{ptr}.\Pi y{:}\texttt{any}. \\ & \texttt{Act}((u_b,u_e).z{:}\texttt{unit}.u_e{>}u_b \wedge \mathsf{Write\ self}\ x\ y\ u_e \\ & \qquad \wedge \mathsf{noActions\ self}@(u_b,u_e), \\ & \quad (u_b,u_e).\top) \\ \mathsf{check}: & \Pi x{:}\texttt{any}. \\ & \texttt{Act}((u_b,u_e).y{:}\texttt{FAE}.u_e{>}u_b \wedge \mathsf{Check\ self}\ x\ y \wedge x = y \\ & \qquad \wedge \mathsf{noActions\ self}@(u_b,u_e), \\ & \quad (u_b,u_e).\top)\end{array}$$

To illustrate, the type for the read action takes a memory location as argument. The partial correctness assertion states that in the time interval (u_b, u_e), this thread performs no actions, and that at time u_e, an atomic read action happens. The type of the returned expression is any, because attackers could (in general) write untyped expressions into memory. The invariant assertion is the trivial formula $\top$. Using the rule ACT, we can type the first action in the interface inc as follows (We omit Σ and Θ for brevity).

$$\begin{array}{l}(u_1,u_2); cnt : \texttt{ptr}; \Delta_1 \vdash \mathsf{read}\ cnt : \\ (x{:}\texttt{any}.u_1{<}u_2 \wedge \mathsf{read\ self}\ cnt\ x\ u_2 \\ \qquad \wedge \mathsf{noAction\ self}@(u_1,u_2), \\ \quad \mathsf{noAction\ self}@(u_1,u_2])\end{array}$$

Rule RET assigns e's type to $\texttt{ret}(e)$. The trace $\mathcal{T}$ containing the complete evaluation of $\texttt{ret}(e)$ satisfies two properties, which appear in the postcondition of $\texttt{ret}(e)$. First, the return expression, which is bound to x, is syntactically equal to e (assertion $(x = e)$). Second, $\mathcal{T}$ satisfies any property φ if φ silent holds. This is because reduction of $\texttt{ret}(e)$ is not effectful. Let the notation $E(\Xi)$ denote t_e and $B(\Xi)$ denote t_b when $\Xi = (t_b, t_e)$. Then, the return expression e is typed at $E(\Xi)$, because e can only be evaluated after the evaluation of $\texttt{ret}(e)$ has ended. The invariant assertion for $\texttt{ret}(e)$ is φ, because before $\texttt{ret}(e)$ returns, the thread is silent.

Rule SEQC types the sequencing statement $\texttt{letc}(c_1, x.c_2)$. The execution of $\texttt{letc}(c_1, x.c_2)$ can be divided into three segments: (1) Silent steps that bring c_1 into evaluation position, (2) computation c_1 runs, and (3) c_2 runs. Segment (1) is necessary for two reasons. First, when a computation is ready to be executed, the thread running it may stay idle for a while to let other programs execute. Second, this thread itself needs a non-effectful transition to push $x.c_2$ onto the stack. The third premise of SEQC splits the time interval Ξ into three connected segments Ξ_0, Ξ_1, and Ξ_2. The logical contexts Δ and its decorated variants contain the ordering constraints of between the end points of these segments. The next three premises of SEQC derive the effect of each of these three segments. The variable contexts in these rules need to include the additional time point parameters that could appear free in Γ or Δs. Because c_2 only executes after c_1 finishes, when type checking c_2, the facts learned from the execution so far (φ_0 and φ_{P1}) are included in the context.

Next, we verify the partial correctness and invariant assertions. For invariant properties, we check that the invariant

assertions for each of the three segments logically entail the invariant property of the sequencing statement. When c_1 is executing, we sequentially compose the effect of the silent segment and the invariant property of c_1. When c_1 finishes and c_2 is executing, we sequentially compose the effect of the silent segment, the partial correctness assertion for c_1, and the invariant assertion for c_2. We substitute the ending time point in the assertions to make the first two segments end at the same time as the sequencing statement. Finally, the last premise of SEQC checks that the partial correctness assertion is logically entailed by the conjunction of the effect of the silent segment and the partial correctness assertions from c_1 and c_2. The type of the return value is the same as that of the second computation.

The following valid typing derivation uses the rule SEQC.

$$(u_b, u_e); cnt : \texttt{ptr}; \Delta_A, \Delta_1 \vdash \\ \texttt{letc}\, x = \mathsf{read}\ cnt; \mathsf{write}\ cnt\ (x{+}1) : (x{:}\texttt{unit}.\varphi_{nd1}, \varphi_{nd1})$$

Here, Δ_A contains axioms about actions, which are shown in Appendix A. We need to discharge several proof obligations to apply the rule SEQC. We show one such obligation below (we omit contexts other than Δ for brevity). Δ_1 is defined in Section IV-C.

$$\Delta_A, \Delta_1 \vdash (\mathsf{noActions\ self@}(u_b, u_1] \wedge \mathsf{noAction\ self@}(u_1, u_2) \\ \wedge\ \mathsf{read\ self}\ cnt\ x\ u_2 \wedge \mathsf{noAction\ self@}(u_2, u_e) \\ \wedge\ \mathsf{write\ self}\ cnt\ (x{+}1)\ u_e) \Rightarrow \varphi_{nd1}(u_b, u_e)$$

B. Expression Typing

Our expression typing judgment $u; \Theta; \Sigma; \Gamma; \Delta \vdash e : \tau$ is parameterized over u, which is the earliest time point at which e is evaluated. Most of the typing rules are standard. A subset is listed in Figure 7.

Rule COMP assigns a monadic type to a suspended computation. Since the suspended computation can only execute after u, the assumption that the beginning time point of c is no earlier than u can be inserted into the logical context. The rule also builds-in weakening of postconditions.

Rule INV allows us to type an expression from the knowledge that it contains no actions. This rule internalizes the intuition behind interface-confinement discussed in Section III-A: if c is a computation that is free of actions and confined to use the interfaces $f_1, \ldots, f_n$ and each of the computations $f_1, \ldots, f_n$ maintains a trace invariant φ while it executes, then as c executes, it maintains φ. The last premise ($\Gamma|_{\texttt{FAE}} \vdash e : \texttt{FAE}$) checks that e is free of actions. There are two ways to derive $e : \texttt{FAE}$. One is to use syntactic derivation rules for $\Gamma \vdash e : \texttt{FAE}$ described in our technical report. The other is to use a dynamic check such as the action check in the example of Section V-A. This action checks syntactically that a closed expression does not contain any action symbols. It returns the expression if the condition is true; otherwise, it gets stuck.

Technically, because φ also accepts as arguments any interval on a trace (it has free variables u_b, u_e), we require that φ be *trace composable*, meaning that if φ holds on two consecutive intervals of a trace, then it hold across the union of the intervals. Formally, $\Gamma.\Xi.\varphi$ is trace composable if given

$$\frac{\Theta; \Sigma; u, \Gamma \vdash \Delta\ \mathsf{ok} \qquad \texttt{fv}(e) \subseteq \texttt{dom}(\Gamma)}{u; \Theta; \Sigma; \Gamma; \Delta \vdash e : \texttt{any}}\ \textsc{Any}$$

$$\frac{\begin{array}{c}\Xi; \Theta; \Sigma; u; \Gamma; \Delta, \Xi \geq u \vdash c : (x{:}\tau.\varphi_1, \varphi_2) \\ \Theta; \Sigma; u, \Xi, \Gamma, x : \tau; \Delta \vdash \varphi_1 \Rightarrow \varphi_1'\ \mathsf{true} \\ \Theta; \Sigma; u, \Xi, \Gamma; \Delta \vdash \varphi_2 \Rightarrow \varphi_2'\ \mathsf{true} \\ \Theta; \Sigma; u, \Gamma \vdash \texttt{comp}(\Xi.x{:}\tau.\varphi_1', \Xi.\varphi_2')\ \mathsf{ok} \\ \texttt{fv}(c) \subseteq \texttt{dom}(\Gamma)\end{array}}{u; \Theta; \Sigma; \Gamma; \Delta \vdash \texttt{comp}(c) : \texttt{comp}(\Xi.x{:}\tau.\varphi_1', \Xi.\varphi_2')}\ \textsc{Comp}$$

$$\frac{\begin{array}{c}\Gamma'.\Xi.\varphi \text{ is trace composable} \\ \Gamma' \subseteq \Gamma \qquad \forall x \in \texttt{dom}(\Gamma'), \Gamma'(x) \text{ is a base type} \\ \Delta' \subseteq \Delta \qquad \Xi; \Theta; \Sigma; u, \Gamma'; \Delta' \vdash \varphi\ \mathsf{silent} \\ \Xi, \Gamma' \vdash \varphi\ \mathsf{ok} \qquad \Gamma|_{\texttt{FAE}} \vdash e : \texttt{FAE}\end{array}}{u; \Theta; \Sigma; \Gamma; \Delta \vdash e : \texttt{inv}(\Xi.\varphi)}\ \textsc{Inv}$$

$$\frac{u; \Theta; \Sigma; \Gamma; \Delta \vdash e : \tau \qquad \tau < \tau'}{u; \Theta; \Sigma; \Gamma; \Delta \vdash e : \tau'}\ \textsc{Sub}$$

$$\frac{\begin{array}{c}u; \Theta; \Sigma; \Gamma; \Delta \vdash e_1 : \texttt{inv}(\Xi.\varphi) \\ u; \Theta; \Sigma; \Gamma; \Delta \vdash e_2 : \texttt{inv}(\Xi.\varphi)\end{array}}{u; \Theta; \Sigma; \Gamma; \Delta \vdash e_1\ e_2 : \texttt{inv}(\Xi.\varphi)}\ \textsc{AppInv}$$

$$\frac{\begin{array}{c}u; \Theta; \Sigma; \Gamma; \Delta \vdash e : \tau \\ \Theta; \Sigma; u, \Gamma; \Delta \vdash e = e'\ \mathsf{true} \qquad \texttt{fv}(e') \subseteq \texttt{dom}(\Gamma)\end{array}}{u; \Theta; \Sigma; \Gamma; \Delta \vdash e' : \tau}\ \textsc{Eq}$$

Fig. 7. Selected expression typing rules

a substitution γ for Γ, three time points t_1 , t_2, and t_3, s.t. $t_1 \leq t_2 \leq t_3$, $(\varphi(t_1, t_2)\gamma \wedge \varphi(t_2, t_3)\gamma) \Rightarrow \varphi(t_1, t_3)\gamma$. Further φ has to hold on intervals when the current thread is silent. This prevents us from derving arbitrary properties of untrusted code. For instance, φ cannot be $\bot$ as no trace can satisfy $\bot$.

The rule INV itself does not stipulate any conditions on the predicate φ, other than requiring that φ be trace composable. However, if e expects some interfaces as arguments, then in applying INV to e, we must choose a φ to match the actual effects of those interfaces, else the application of e to the interfaces cannot be typed.

Expressions of type $\texttt{inv}(\Xi.\varphi)$ are commonly used to type interface-confined adversary-supplied code, which can use expressions in untyped ways. Rule APPINV allows an expression e_1 of type $\texttt{inv}(\Xi.\varphi)$ to be applied to another expression of the same type. The resulting application has the same invariant type. Note that e_2 has to have an invariant type, as e_1 could use e_2 in untyped ways.

We also define a subtyping relation that allows well-typed terms, to be assigned an invariant type. Most rules of this relation are standard. We show some of these rules below.

$$\frac{}{\Pi x{:}\texttt{inv}(\Xi.\varphi).\texttt{inv}(\Xi.\varphi) < \texttt{inv}(\Xi.\varphi)}\ \textsc{Inv1}$$

$$\frac{\tau < \texttt{inv}(\Xi.\varphi)}{\texttt{comp}(\Xi.x{:}\tau.\varphi, \Xi.\varphi) < \texttt{inv}(\Xi.\varphi)}\ \textsc{Inv2} \qquad \frac{}{\texttt{b} < \texttt{inv}(\Xi.\varphi)}\ \textsc{Base}$$

Rule INV1 states that a function type is a subtype of an invariant type if both the argument and result of the function are of that invariant type. This is sound because the semantics of the function type are stricter than the semantics of the invariant type. Similarly, INV2 allows a computation type to be a subtype of an invariant type, if the effects of that computation satisfy the invariant, and the return type of the computation is the same invariant type. However, the converse of these two rules does not hold. The reason is that terminating expressions that have function (computation) types evaluate to functions (suspended computations), while terminating expressions of the invariant type may evaluate to any stuck terms.

Now we can type the trusted program called c in our example (Figure 2). We explain how to type the sub-expression $y\ inc\ double\ prn$. First, applying the COMP rule, we derive that inc has type $\texttt{comp}((u_b, u_e).x{:}\texttt{unit}.\varphi_{ndI}, (u_b, u_e).\varphi_{ndI})$. Then using the subtyping relation (rules BASE and INV2), we can show that inc has the type $\texttt{inv}((u_b, u_e).\varphi_{ndI})$. The other two interfaces can be typed similarly. We apply INV1 in typing prn, which we can do because prn takes an interface of type $\texttt{inv}((u_b, u_e).\varphi_{ndI})$ as an argument. Next, the postcondition of the check action adds to the typing context y : FAE, which matches the last premise of INV. $\varphi_{ndI}(u_b, u_e)$ satisfies the other premises of INV, so we can conclude that y : $\texttt{inv}((u_b, u_e).\varphi_{ndI})$. Applying the rule APPINV three times, we finally derive that $y\ inc\ get\ prn$ has type $\texttt{inv}((u_b, u_e).\varphi_{ndI})$.

Our last typing rule, EQ, assigns an expression e', the type of e, if e is syntactically equal to e'. This rule is useful for typing programs read from adversary-modifiable memory locations when separate reasoning can establish that the value stored in the location is, in fact, syntactically equal to some known expression with a known type. Depending on the application, such reasoning may be based on a dynamic check (e.g., in secure boot [12] the hash of a textual reification of a program read from adversary-accessible memory is compared to the corresponding hash of a known program before executing the read program) or it may be based on a logical proof showing the inability of the adversary to write the location in question (e.g., showing that guests cannot write to hypervisor memory). We show a concrete example in Section VII.

C. Soundness

We prove our type system sound relative to the semantic model of Section IV-B. For instance, the soundness theorem for expression typing states that if e is typed τ using the syntactic typing rules under relevant typing contexts, then e is in the semantic interpretation of τ with proper substitutions for those typing contexts. We first define valid substitutions for contexts. We write $\mathcal{RT}[\![\Theta]\!]$ to denote the set of valid semantic substitutions for Θ, which map type variables to C. We write $\mathcal{RG}[\![\Gamma]\!]^{u}_{\theta;\mathcal{T}}$ to denote a set of substitutions for variables in Γ. Each indexed substitution is a pair of an index and a substitution γ for variables.

Theorem 2 (Soundness)**.**
If $\forall A :: \alpha \in \Sigma$, $\forall \mathcal{T}, \mathcal{K}, \iota, k, (k, A) \in \mathcal{RA}[\![\alpha]\!]^{\mathcal{K},\iota}_{\cdot;\mathcal{T}}$, *then*

1) $u; \Theta; \Sigma; \Gamma; \Delta \vdash e : \tau$, $\forall \theta \in \mathcal{RT}[\![\Theta]\!]$, $\forall t, t'$, $t' \geq t$, *let* $\gamma_u = [t/u]$, $\forall \mathcal{T}$, $\forall k, \gamma$, $(k, \gamma) \in \mathcal{RG}[\![\Gamma\gamma_u]\!]^{t'}_{\theta;\mathcal{T}}$, $\mathcal{T} \vDash \Delta\gamma\gamma_u$ *implies* $(k, e\gamma) \in \mathcal{RE}[\![\tau\gamma\gamma_u]\!]^{t'}_{\theta;\mathcal{T}}$
2) $\Xi; \Theta; \Sigma; \Gamma; \Delta \vdash c : \eta$, $\forall\ \mathcal{K}$, ι, *let* $\gamma_\Xi = [\mathcal{K}, \iota/\Xi, \mathsf{self}]$ $\forall \theta \in \mathcal{RT}[\![\Theta]\!]$, $\forall \mathcal{T}$, $\forall k, \gamma, (k, \gamma) \in \mathcal{RG}[\![\Gamma\gamma_\Xi]\!]^{B(\mathcal{K})}_{\theta;\mathcal{T}}$, $\mathcal{T} \vDash \Delta\gamma\gamma_\Xi$ *implies* $(k, c\gamma) \in \mathcal{RC}[\![\eta\gamma\gamma_\Xi]\!]^{\mathcal{K},\iota}_{\theta;\mathcal{T}}$
3) $\Theta; \Sigma; \Gamma; \Delta \vdash \varphi$ true, $\forall\ t\ \forall \theta \in \mathcal{RT}[\![\Theta]\!]$, $\forall \mathcal{T}$, $\forall k, \gamma, (k, \gamma) \in \mathcal{RG}[\![\Gamma]\!]^{t}_{\theta;\mathcal{T}}$, $\mathcal{T} \vDash \Delta\gamma$ *implies* $\mathcal{T} \vDash \varphi\gamma$

Theorem 2 is proved by induction on typing derivations and a subinduction on step-indices for the case of fixpoints. The theorem assumes the soundness of the action specifications in Σ, which we explain in our technical report.

An immediate corollary of the soundness theorem is the following robust safety theorem, which states that the invariant assertion of a computation c's postcondition holds even when c executes concurrently with other threads, which may be adversarial. The theorem holds because we account for adversarial actions in the definition of $\mathcal{RC}[\![\eta]\!]$. A similar theorem holds for partial correctness assertions.

Theorem 3 (Robust safety)**.** *If* $u_1, u_2; \Delta \vdash c : (\_, \varphi)$, $\mathcal{T} \vDash \Delta$, *$\mathcal{T}$ is a trace obtained by executing the parallel composition of threads of ID (ι_1, .. ι_k), at time t_b, the computation that thread ι_j is about to run is c, and at time t_e, c has not returned, then* $\mathcal{T} \vDash \varphi[t_b, t_e, \iota_j/u_1, u_2, \mathsf{self}]$.

D. Discussion

Call-by-name reduction The use of call-by-name reduction (instead of call-by-value reduction) simplifies our system significantly. If we were to use a call-by-value semantics, then the soundness of the function application rule would require all predicates to be closed under beta reduction, which would rule out syntactic equality from our assertion logic. If we were to use beta-equality in the assertion logic instead of syntactic equality, then we would not be able to prove the EQ rule sound using step-indexing because beta-equality does not consider the number of reduction steps.

Interface-confinement Our interface-confinement check is purely syntactic—we rely on the syntactic absence of state-accessing actions in adversarial code. In many cases, interface-confinement may not be obvious from syntactic restrictions and it may be based on the impossibility of the adversarial code ever obtaining references to sensitive code or data. This is common in object capability systems, which System M cannot model in full generality. Nonetheless, we consider our design pragmatic because our syntactic check is very easy to implement and it can encode capability-based protection in many cases.

Interface-confined adversarial code is very similar to unknown client code (or quantification over evaluation contexts), which proof and type systems for program verification often consider. The main difference is that in most work (with the exception of formal systems for reasoning about protocols such as F7 [7] and Dupressoir et al.'s work [13]), clients of abstract data types are typed; clients will not evaluate to

stuck terms such as (true 3). In contrast, our INV rule allows reasoning about untyped client programs. (This rule can be applied to reasoning about typed client code as well.) INV is absent from most existing reasoning systems, as they do not need to consider dynamically obtained untrusted code (e.g., code read from untyped memory, or code received over the network).

VI. Composition and Rely-Guarantee Reasoning

The typing rules introduced so far focus on one executing program. When reasoning about distributed systems, we wish to compose properties that are derived from typing known programs to prove properties of the whole system. In the following, we present System M's principles for compositional reasoning with multiple programs. Even though variants of these principles have been introduced in prior first-order systems [14], [6], we discuss them here as they are crucial to reasoning in System M.

A. Extended example

We motivate these principles by expanding our example from Section III-A to allow multiple programs to modify the counter. We add locks to protect memory locations. In the shared state, an additional data structure L maps each memory location to the ID of the thread that holds the location's lock. Only the thread, whose ID is $L(l)$ can read and write the location l (this is enforced by the operational semantics). Consider two threads ι_1 and ι_2 that sit in an infinite loop. In each iteration, each first uses the interfaces shown in Figure 2 to access the counter, then yields the lock of cnt to the other thread. We show the programs below. Thread ι_1 runs c_1 and ι_2 runs c_2.

F = fix $f(i) =$ comp(letc $x =$ download();
letc $y =$ check x;
y *inc double prn*;
yieldTo cnt i
lete _ $= f(i)$; ret())
c_1 = lete _ = F ι_2; ret()
c_2 = lete _ = F ι_1; ret()

Assume that, initially, ι_1 holds the lock. We would like to show the invariant that the counter never decreases, i.e., $\varphi_{nd}(0,\infty)$ holds. Since ι_1 and ι_2 share the counter, reasoning about either one of them is not enough. We need reasoning principles to incorporate the effects of one program while reasoning about another program.

B. Reasoning about Global Configurations

The assertions derived using System M's typing rules are *local properties* of a program. The soundness requirement prevents us from specifying a postcondition of the read action that depends on other threads' or self's actions outside the specific time interval in which the action happens. Because these properties are local to both the time interval and the thread, we can compose the property of a program both sequentially with other computations within one thread, and in parallel with other threads. We have already seen sequential composition in the typing rules for sequencing statements, where the effects of the sequential composition of c_1 and c_2 are simply the conjunction of the effects of c_1 and c_2. In the following, we develop rely-guarantee style reasoning principles to compose local properties of programs executing concurrently.

First, System M includes a rule called Honest that internalizes the invariant of a program into a logical formula, which subsequently facilitates its composition with properties of other programs.

$$\frac{\begin{array}{c}\Xi;\Theta;\Sigma;\Gamma;\Delta \vdash c : (\_,\varphi) \\ \Theta;\Sigma;\Gamma;\Delta \vdash \mathsf{start}(i,\mathtt{comp}(c),t)\ \mathsf{true} \\ \Theta;\Sigma \vdash \Gamma\ \mathsf{ok} \qquad \forall x \in \mathtt{dom}(\Gamma), \Gamma(x) \text{ is a base type}\end{array}}{\begin{array}{c}\Theta;\Sigma;\Gamma;\Delta \vdash \forall u{:}\mathtt{time}.u \geq t \Rightarrow \\ \varphi[(t,u), i/\Xi, \mathsf{self}]\ \mathsf{true}\end{array}}\ \textsc{Honest}$$

In words: If we know that a thread i starts executing c at time t (premise $\mathsf{start}(i,\mathtt{comp}(c),t)$) and computation c has an invariant postcondition φ, then we can conclude that at any later point u, φ holds for the interval $(t,u]$. The condition that c be typed under a Γ that maps variables to only base types, is required by the soundness proofs. To prove this rule sound, we need to prove that given any substitution (n,γ) for Γ, the trace satisfies the invariant φ of c. From the second premise of Honest, we know that c starts with an empty stack. c can never return because there is no frame to be popped off the empty stack. Therefore, at any time point after c starts, the invariant of c should hold. However, the length of the trace after c starts, denoted m, is not related to n. To use the induction hypothesis, we need to use substitution (m,γ) for Γ. The interpretation of base types has the property that if (n,e) is in the interpretation of the base type τ, then (m,e) is in the interpretation of τ. This completes the proof.

Next, System M includes a simplified version of the rely-guarantee reasoning principle described by Garg et al. [6] to prove invariant properties of the form $\forall t{:}\mathtt{time},\ \varphi(t)$. We list the conditions below.

($\mathbf{RG}_1$) $\varphi(t_i)$
($\mathbf{RG}_2$) $\forall u, (\forall u', t_i{<}u'{<}u \Rightarrow \varphi(u')) \Rightarrow (\forall i, \zeta(i) \Rightarrow \psi(i,u))$
($\mathbf{RG}_3$) $\forall u, (\forall u', t_i{<}u'{<}u \Rightarrow \varphi(u')) \Rightarrow (\forall i, \zeta(i) \Rightarrow \psi(i,u)) \Rightarrow \varphi(u)$

The following RG rule is used to prove invariant properties.

$$\frac{\begin{array}{c}\Theta;\Sigma;\Gamma;\Delta \vdash \mathbf{RG}_1\ \mathsf{true} \\ \Theta;\Sigma;\Gamma;\Delta \vdash \mathbf{RG}_2\ \mathsf{true} \qquad \Theta;\Sigma;\Gamma;\Delta \vdash \mathbf{RG}_3\ \mathsf{true}\end{array}}{\Theta;\Sigma;\Gamma;\Delta \vdash \forall u{:}\mathtt{time}, u \geq t_i \Rightarrow \varphi(u)\ \mathsf{true}}\ \text{RG}$$

Informally, to prove that a global invariant $\varphi(u)$ holds at all times u after an initial time t_i, select a relevant set of threads which affect the state that the invariant talks about (this set is defined by the predicate ζ) and for each relevant thread, select an invariant $\psi(i,u)$. Now prove the following three properties. ($\mathbf{RG}_1$): $\varphi(t_i)$ holds; ($\mathbf{RG}_2$): if φ holds at all time points strictly less than u, then $\psi(i,u)$ holds for each $i \in \zeta$; and ($\mathbf{RG}_3$):

assuming that $\psi(i,u)$ holds for each $i \in \zeta$, $\varphi(u)$ holds. One may think of $\psi(i,u)$ as the local guarantee of the thread i and $\varphi(u)$ as a global "rely". Then, this rule is strongly reminiscent of standard rely-guarantee reasoning. The rule can be proved sound by induction over time points. For the rest of this paper, we call $\varphi(u)$ a global invariant, and $\psi(i,u)$ a local guarantee by thread i.

Using the rule RG, we show that *cnt*'s value never decreases in our extended example. Suppose that the system starts at time 0. Assumptions about the system's initial state are: $\Delta_2 = \mathsf{lock}\ cnt\ \iota_1\ l\ 0$ (the lock on l is initially held by ι_1), $\mathsf{start}\ \iota_1\ (\mathtt{comp}(c_1))\ 0$ (thread ι_1 executes program c_1 starting at time 0), $\mathsf{start}\ \iota_2\ (\mathtt{comp}(c_2))\ 0$ and $\forall i, \mathsf{noActions}\ i\ 0$ (no actions are performed by any thread at time 0). Using rule RG, we prove a strong global invariant.

$$\varphi_{nd2}(u) = 0 \leq u \Rightarrow (\mathsf{lock}\ cnt\ \iota_1\ u \vee \mathsf{lock}\ cnt\ \iota_2\ u) \wedge \varphi_{nd}(0,u)$$

The local guarantee is defined below:

$$\begin{aligned}\psi(i,t) = &\forall x, \mathsf{yieldTo}\ i\ x\ t \Rightarrow \exists y, \mathsf{eval}\ x\ y \wedge \{i,y\} \subseteq \{\iota_1,\iota_2\} \\ &\wedge \forall z, y, \mathsf{write}\ i\ z\ y\ t \Rightarrow \\ &\quad \exists t', l, v, v', t' < t \wedge \mathsf{eval}\ z\ l \wedge \mathsf{eval}\ y\ v \\ &\quad\quad \wedge \mathsf{mem}\ l\ v'\ t' \wedge \mathsf{noWrite}\ l@(t',t) \wedge v' \leq v\end{aligned}$$

The local guarantee states that thread i will yield to ι_1, if i is ι_2, and to ι_2, if i is ι_1, and that if thread i writes to a memory location, then the value that i writes is greater than or equal to the value that was most recently written to that location. The thread selector is defined as $\zeta(i) = (i = \iota_1) \vee (i = \iota_2)$.

Given the new semantics that a lock is required for a thread to read or write a memory location, we include axioms stating that if a read (write) action is successful, then that thread must have the lock at the time of the read (write).

Next, we show that $\mathbf{RG}_1$, $\mathbf{RG}_2$, and $\mathbf{RG}_3$ hold. $\mathbf{RG}_1$ can be proved directly using the assumptions in Δ_2. $\mathbf{RG}_3$ requires that local guarantees of ι_1 and ι_2 imply the global invariant. We can directly use first-order logic rules to prove $\mathbf{RG}_3$, assuming Δ_2. We elide the details here.

To prove $\mathbf{RG}_2$, we first type c_1. We use the principles for reasoning about interface-confined code described in Section V. The invariant we show for c_1 is:

$$\mathtt{inv}((u_b,u_e).\mathsf{self} = \iota_1 \Rightarrow \psi[\mathsf{self}/i]@(u_b,u_e])$$

Using the HONEST rule, we derive:

$$cnt : \mathtt{ptr}; \Delta_A, \Delta_2 \vdash \forall u, u > 0 \Rightarrow \mathsf{self} = \iota_1 \Rightarrow \psi(\iota_1)@(0,u)$$

Similarly, we can prove

$$cnt : \mathtt{ptr}; \Delta_A, \Delta_2 \vdash \forall u, u > 0 \Rightarrow \mathsf{self} = \iota_2 \Rightarrow \psi(\iota_2)@(0,u)$$

Combing the two, we get

$$\begin{aligned}cnt : \mathtt{ptr}; \Delta_A, \Delta_2 \vdash \forall u, i, u > 0 \Rightarrow (i = \iota_1) \vee (i = \iota_2) \\ \Rightarrow \psi(i)@(0,u)\end{aligned}$$

$\mathbf{RG}_2$ follows immediately. Finally, the rule RG yields $\forall t, \varphi_{nd2}(t)$.

```
1  runmodule(srvc, snap, req, Nloc) =
2    ...
3    extend_pcr(pcr17, srvc);
4    letc s = check srvc;
5    letc (skey, freshness_tag) = NVRAMread Nloc;
6    lete state = check_decrypt_snapshot (snap);
7    letc new_tag = hash (freshness_tag||req);
8    NVRAMwrite(Nloc, (new_summary, skey));
9    extend_pcr(pcr17, 0);
10   letc (state', resp)
         = (s ExtendPCR ResetPCR ...) (state, req);
11   ...
```

Fig. 8. Snippet of invocation code

VII. CASE STUDY

Memoir is a system that uses a Trusted Platform Module (TPM) to implement security-sensitive services [10]. We modeled Memoir-Basic, a key subset of the features of Memoir, and verified its main security property by hand in System M. We briefly outline this development here. We first review Memoir and then highlight the use of System M's novel typing rules in our proof.

A. Overview of Memoir

Memoir provides state-integrity guarantees for stateful security-sensitive services invoked by potentially malicious parties. Such services often rely on untrusted storage to store their persistent state. An example of such a service is a password manager that responds with a stored password when it receives a request containing a URL and a username. In addition to the secrecy and integrity of its state, the service would also need to prevent the attacker from invoking the service with a valid but old state, and consequently mounting service rollback attacks. Simply encrypting and signing the service's state in persistent storage cannot prevent such attacks. Memoir solves this problem by using the TPM to provide state integrity guarantees. Memoir relies on the following TPM features: PCRs, late launch, and non-volatile RAM or NVRAM [1].

Memoir has two phases: service initialization and service invocation. During initialization, the Memoir module is assigned an NVRAM block. It is also given a service to protect. The module generates a new symmetric key that is used throughout the lifetime of the service. It sets the permissions on accesses to the NVRAM block to be tied to the hash stored in PCR 17, which contains the hash of the code for Memoir and the service. To prevent rollback attacks, it uses a *freshness tag* which is a chain of hashes of all the requests received so far. The secret key and an initial freshness tag are stored in the designated NVRAM location. The service then runs for the first time to generate an initial state, which along with the freshness tag is encrypted with the secret key and stored on disk. This encryption of the service's state along with the freshness tag is called a *snapshot*.

After initialization, a service can be invoked by providing Memoir with an NVRAM block $Nloc$, code for the service $srvc$, and a snapshot $snap$. In Figure 8, we show a snippet of the Memoir service invocation code, called $runmodule$. On line 3, PCR 17 is extended with the code for service. On line 4, a syntactic check ensures that the service has no free actions. On line 5, Memoir retrieves the key and freshness tag from the NVRAM. On lines 6-7, Memoir decrypts the snapshot and verifies that the freshness tag in the provided state matches the one stored in NVRAM. If the verification succeeds, Memoir computes a new freshness tag and updates the NVRAM. Next, on line 8, it executes the service to generate a new state and a response. The new snapshot corresponding to the new state and freshness tag is stored on disk.

B. Proof outline

The security property we prove about Memoir is that the service can only be invoked on the state generated by the last completed instance of the service. The proof proceeds by proving the following three invariants about $runmodule$:

- **PCR Protection (φ_{in1}):** The value of PCR 17 contains a certain hash h only during a late launch session running $runmodule$. The hash h is the late launch signature of $runmodule$ extended with $srvc$.
- **NVRAM Protection (φ_{in2}):** After the permissions on the NVRAM have been set to be tied to h, the permissions on that location are never changed.
- **Key Secrecy (φ_{in3}):** If the key corresponding to the service is available to a thread, then the thread must have either generated it or read it from the NVRAM.

Reasoning about adversary-supplied code Each of the three global invariants is proved using the rules RG and HONEST. The code $runmodule$ is typed with an invariant postcondition corresponding to each global invariant. The crucial step in typing $runmodule$ is typing line 10, where the service s, provided by the adversary, is executed. When s is executed, it takes as arguments interface functions corresponding to atomic actions in our model and TPM interfaces. (Our technical report contains a complete list of the actions in our model.) Shown in Figure 8 are the two TPM interfaces for extending ($ExtendPCR$) and resetting PCR ($ResetPCR$).

We now describe for each of the invariants: φ_{in1}, φ_{in2}, and φ_{in3}, how s applied to interfaces is typed to preserve that invariant.

Derivation of φ_{in1}: We show that s cannot exit the late launch session with the state of PCR 17 being a prefix of h. As a post-condition of line 10, we know that PCR 17 has been extended with 0. Each of the functions in the interface provided to s, ensures that the value of PCR 17 cannot be reversed to be h, which can only be generated by a late launch call on $runmodule$. Therefore, by the INV rule, we show the same for $(s\ ExtendPCR\ ResetPCR\ \cdots)$.

Derivation of φ_{in2}: As with the previous case, since the value of PCR17 has been extended with 0, no interface provided to s allows changing the permissions on the NVRAM location, and we use the INV rule to assert the same property for $(s\ ExtendPCR\ ResetPCR\ \cdots)$.

Derivation of φ_{in3}: Once we can prove that the permissions on $Nloc$ are always tied to PCR 17 being h, by $Nloc$'s access control mechanism, we infer that the current state of PCR 17 must be identical to h and therefore as h includes the hash of the service Memoir was initialized with, we have $srvc = service$ (where $=$ denotes syntactic equality). Here, $service$ is the service that Memoir was initialized with. We make an assumption that $service$ does not leak the secret key when applied to the interfaces $(ExtendPCR\ ResetPCR\ \cdots)$. (This property could be verified either by manual audits or automated static analysis of the service code). We then use the EQ rule to assign the same condition to s.

C. Comparison to TLA+ Proof of Memoir

The original paper on Memoir [10] is accompanied by a machine-checked proof of state integrity in TLA+ [15]. State integrity is specified as a high-level, ideal transition system that does not allow the state to be rolled back or tampered with. Memoir is modeled as a much more detailed, low-level transition system. Security is proved by showing that the low-level transition system simulates the ideal transition system.

We find that their low-level model of Memoir does not include several implementation details that are required by our proofs. In fact, it is possible to derive straightforward attacks on Memoir in the absence of these details. This is because their model makes the simplifying modeling choice that the service being protected is a constant, effect-free predicate. We adopt a more realistic model of the service. The service in our model is code provided by the adversary, which runs in the same privileged late launch session as Memoir and is only confined by the interfaces provided by the TPM hardware. The more realistic treatment of the protected service necessitates the following details in our modeling of Memoir:

- On line 9 of $runmodule$, we need to extend the hash in PCR 17 with some value. The postcondition of this line allows us to prove φ_{in1} and φ_{in2}. If the hash in PCR 17 is not extended before calling the service, and if the service exits the late launch session, other programs are free to read from the NVRAM location since the value of PCR 17 matches the permission on the NVRAM location.
- On line 3, we extend the hash in PCR 17 with code for the service. This allows us to prove that the code for the service does not change between invocations.
- Finally, we need to assume that the code for the service that Memoir is initialized with does not leak the secret key, which is a possibility since the service runs in the same privileged late launch session as Memoir. We use this assumption in the proof for φ_{in3}, as an antecedent to the EQ rule.

VIII. RELATED WORK

Hoare Type Theory (HTT) In HTT [16], [17], [18], a monad classifies effectful computations, and is indexed by the return type, a pre-condition over the (initial) heap, and

a postcondition over the initial and final heaps. This allows proofs of functional correctness of higher-order imperative programs. The monad in System M is motivated by, and similar to, HTT's monad. However, there are several differences between System M's monad and HTT's monad. A System M postcondition is a predicate over the entire execution trace, not just the initial and final heaps as in HTT. It also includes an invariant assertion which holds even if the computation does not return. This change is needed because we wish to prove safety properties, not just properties of heaps. Although moving from predicates over heaps to predicates over traces in a sequential language is not very difficult, our development is complicated because we wish to reason about robust safety, where adversarial, potentially untyped code interacts with trusted code. Hence, we additionally incorporate techniques to reason about untyped code (rules EQ and INV).

RHTT [19] is a relational extension of HTT used to reason about access control and information flow properties of programs. That extension to HTT is largely orthogonal to ours and the two could potentially be combined into a larger framework. The properties that can be proved with RHTT and System M are different. System M can verify safety properties in the presence of untyped adversaries; RHTT verifies relational, non-trace properties assuming fully typed adversaries.

LS^2 and PCL System M is inspired by and based upon a prior program logic, LS^2, for reasoning about safety properties of first-order order programs in the presence of adversaries [6]. The main conceptual difference from LS^2 is that in System M trusted and untrusted components may exchange code and data, whereas in LS^2 this interface is limited to data. Our INV rule for establishing invariants of an unknown expression from invariants of interfaces it has access to is based on a similar rule called RES in LS^2. The difference is that System M's rule allows typing higher-order expressions, which makes it more complex. LS^2 itself is based on a logic for reasoning about Trusted Computing Platforms [20] and Protocol Composition Logic (PCL) for reasoning about safety properties of cryptographic protocols [14].

Type systems that reason about adversary-supplied code The idea of using a non-informative type, `any`, for typing expressions obtained from untrusted sources goes back to the work of Abadi [21]. Gordon and Jeffrey develop a very widely used proof technique for proving robust safety based on this type [22]. In their system, any program can be *syntactically* given the type `any` by typing all subexpressions of the program `any`. Although System M's use of the `any` type is similar, our proof technique for robust safety is different. It is *semantic* and based on that in PCL—we allow for arbitrary adversarial interleaving actions in the semantics of our computation types (relation $\mathcal{RC}[\![\eta]\!]$ in Section IV). Due to this generalized semantic definition, robust safety (Theorem 3) is a trivial consequence of soundness (Theorem 2).

Several type systems for establishing different kinds of safety properties build directly or indirectly on the work of Abadi [21] and Gordon and Jeffrey [22]. Prominent among these are RCF [23] and its extensions [7], [24]. RCF is based on types refined with logical assertions, which provide roughly the same expressiveness as System M's dependently-typed computation types. By design, RCF's notion of trace is monotonic: the trace is an unordered *set* of actions (programmer-specified ghost annotations) that have occurred in the past [25]. This simplified design choice allows scalable implementation. On the other hand, there are safety properties of interest that rely on the order of past events and, hence, cannot be directly represented in RCF's limited model of traces. An example of this kind is measurement integrity in attestation protocols [20, Theorems 2 & 4]. In contrast to RCF, we designed System M for verification of general safety properties (so the measurement integrity property can be expressed and verified in System M), but we have not considered automation for System M so far.

F* [24] extends F7 with quantified types, a rich kinding system, concrete refinements and several other features taken from the language Fine [26]. This allows verification of stateful authorization and information flow properties in F*. Quantified predicates can also be used for full functional specifications of higher-order programs. Although we have not considered these applications so far, we believe that System M can be extended similarly.

The main novelty of System M compared to the above-mentioned line of work lies in the EQ and INV rules that statically derive computational effects of untyped adversary-supplied code.

Code-Carrying Authorization (CCA) [27] is another extension to [22] that enforces authorization policies. CCA introduces dynamic type casts to allow untrusted code to construct authorization proofs (e.g., Alice can review paper number 10). The language runtime uses logical assertions made by trusted programs to constructs proofs present in the type cast. The soundness of type cast in CCA relies on the fact that untrusted code cannot make any assertions and that it can only use those made by trusted code. Similar to CCA, System M also assigns untrusted code descriptive types. CCA checks those types at runtime; whereas the INV rule assigns types statically.

Verification of the TPM and protocols based on the TPM Existing work on verification of TPM APIs and protocols relying on TPM APIs uses a variety of techniques [28], [29], [30], [31], [20]. Gurgens et al. uses automata to model the transitions of TPM APIs [31]. Several results [28], [29], [30] use the automated tool Proverif [32]. Proverif over-approximates the protocol states and works with a monotonic set of facts. Special techniques need to be applied to use Proverif to analyze stateful protocols such as ones that use TPM PCRs [28]. System M is more expressive: it can model and reason about higher-order functions and programs that invoke adversary-supplied code. Reasoning about shared non-monotonic state is possible in System M. However, verification using System M requires manual proofs. It is unclear whether our Memoir case study can be verified using the techniques

introduced in [28], as it requires reasoning about higher-order code.

A proof of safety formalized in TLA+ [15] was presented in the Memoir paper [10]. Our model is more detailed and our manual proof reveals subtle assumptions made by the TLA+ proof. A comprehensive comparison is in Section VII-C.

IX. Conclusion

System M combines invariants with an intuitive principle to reason about the effects of interface-confined untrusted code that executes without runtime monitoring and without deep type analysis. System M facilitates reasoning about trace properties of programs that interact with the adversary by sending first-order messages, through shared state, and through execution of attacker-supplied code. We provide sound reasoning principles for interface-confinement: the invariants of interface-confined code can be established from the invariants of the interfaces available to it. Our Memoir case study reveals critical assumptions that the security of Memoir relies on.

Acknowledgement

This research was supported in part by AFOSR MURI award no. FA9550-11-1-0137, NSA under NSA/CMU SoS Lablet, and NSF grant CNS1018061.

References

[1] TrustedComputingGroup, "TPM library specification," http://www.trustedcomputinggroup.org/resources/tpm_library_specification.
[2] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE Trans. Software Eng.*, vol. 3, no. 2, pp. 125–143, 1977.
[3] A. Taly, . Erlingsson, J. C. Mitchell, M. S. Miller, and J. Nagra, "Automated analysis of security-critical JavaScript APIs." in *IEEE Symposium on Security and Privacy*, 2011.
[4] "ADsafe," onlein at http://www.adsafe.org/.
[5] A. Nanevski, G. Morrisett, and L. Birkedal, "Hoare type theory, polymorphism and separation," *Journal of Functional Programming*, vol. 18, no. 5&6, pp. 865–911, 2008.
[6] D. Garg, J. Franklin, D. Kaynar, and A. Datta, "Compositional system security in the presence of interface-confined adversaries," *Electronic Notes in Theoretical Computer Science*, vol. 265, pp. 49–71, 2010.
[7] K. Bhargavan, C. Fournet, and A. D. Gordon, "Modular verification of security protocol code by typing," in *Proc. POPL*, 2010.
[8] G. C. Necula, "Proof-carrying code," in *Proc. POPL*, 1997.
[9] A. Ahmed, "Step-indexed syntactic logical relations for recursive and quantified types," in *Proc. ESOP*, 2006.
[10] B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. McCune, "Memoir: Practical state continuity for protected modules," in *Proc. IEEE S&P*, 2011.
[11] L. Jia, S. Sen, D. Garg, and A. Datta, "A logic of programs with interface-confined code," Carnegie Mellon University, Tech. Rep. CMU-CyLab-13-001, 2013.
[12] B. Parno, J. M. McCune, and A. Perrig, "Bootstrapping trust in commodity computers," in *Proc. S&P*, 2010, pp. 414–429.
[13] F. Dupressoir, A. D. Gordon, J. Jurjens, and D. A. Naumann, "Guiding a general-purpose c verifier to prove cryptographic protocols," in *Proceedings of the 2011 IEEE 24th Computer Security Foundations Symposium*, ser. CSF '11, 2011.
[14] A. Datta, A. Derek, J. C. Mitchell, and A. Roy, "Protocol Composition Logic (PCL)," *Electronic Notes in Theoretical Computer Science*, vol. 172, pp. 311–358, 2007.
[15] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.
[16] A. Nanevski, G. Morrisett, and L. Birkedal, "Hoare type theory, polymorphism and separation," *Journal of Functional Programming*, vol. 18, no. 5&6, pp. 865–911, 2008.
[17] A. Nanevski, A. Ahmed, G. Morrisett, and L. Birkedal, "Abstract predicates and mutable ADTs in Hoare type theory," in *Proc. ESOP'07*, 2007.
[18] A. Nanevski, P. Govereau, and G. Morrisett, "Towards type-theoretic semantics for transactional concurrency," in *Proc. TLDI'09*, 2009.
[19] A. Nanevski, A. Banerjee, and D. Garg, "Verification of information flow and access control policies via dependent types," in *Proc. IEEE S&P*, 2011.
[20] A. Datta, J. Franklin, D. Garg, and D. Kaynar, "A logic of secure systems and its application to trusted computing," in *Proc. IEEE S&P*, 2009.
[21] M. Abadi, "Secrecy by typing in security protocols," *Journal of the ACM*, vol. 46, no. 5, 1999.
[22] A. D. Gordon and A. Jeffrey, "Authenticity by typing for security protocols," *Journal of Computer Security*, vol. 11, no. 4, pp. 451–519, Jul. 2003.
[23] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis, "Refinement types for secure implementations," *TOPLAS*, vol. 33, no. 2, pp. 8:1–8:45, 2011.
[24] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang, "Secure distributed programming with value-dependent types," in *Proc. ICFP*, 2011.
[25] C. Fournet, A. D. Gordon, and S. Maffeis, "A type discipline for authorization policies," *TOPLAS*, vol. 29, no. 5, 2007.
[26] N. Swamy, J. Chen, and R. Chugh, "Enforcing stateful authorization and information flow policies in fine," in *Proc. ESOP*, 2010.
[27] S. Maffeis, M. Abadi, C. Fournet, and A. D. Gordon, "Code-carrying authorization," in *Proc. ESORICS '08*, 2008.
[28] S. Delaune, S. Kremer, M. D. Ryan, and G. Steel, "Formal analysis of protocols based on TPM state registers," in *Proc. CSF'11*, 2011.
[29] ——, "A formal analysis of authentication in the TPM," in *Proc. FAST'10*, 2011.
[30] L. Chen and M. Ryan, "Attack, solution and verification for shared authorisation data in TCG TPM," in *Proc. FAST'09*, 2010.
[31] S. Gürgens, C. Rudolph, D. Scheuermann, M. Atts, and R. Plaga, "Security evaluation of scenarios based on the TCG's TPM specification," in *Proc. ESORICS'07*, 2007.
[32] B. Blanchet, "Using Horn clauses for analyzing security protocols," in *Formal Models and Techniques for Analyzing Security Protocols*, ser. Cryptology and Information Security Series, V. Cortier and S. Kremer, Eds., Mar. 2011, vol. 5, pp. 86–112.

Appendix

A. Axioms used in Examples

We list axioms that we use in the typing derivations for the example programs here. First, we define predicate $\mathsf{noWrite}\, l\ t$ to mean that at time t, location l is not written by any thread.

$$\mathsf{noWrite}\ l\ t = \forall i, x, y, l', \mathsf{write}\ i\ x\ y \wedge \mathsf{eval}\ x\ l' \wedge l = l' \Rightarrow \bot$$

$$\begin{array}{ll}
A1 & \forall i, e, v, t, \mathsf{read}\ i\ e\ v\ t \Rightarrow \exists l, \mathsf{eval}\ e\ l \wedge \mathsf{mem}\ l\ v\ t \\
A2 & \forall l, v, t, v', t', \mathsf{mem}\ l\ v\ t \wedge \mathsf{mem}\ l\ v'\ t' \wedge \\
 & \qquad \mathsf{noWrite}\ l@(t, t'] \Rightarrow v = v' \\
A3 & \forall i, e, e', t, \mathsf{write}\ i\ e\ e'\ t \Rightarrow \\
 & \quad \exists l, v, \mathsf{eval}\ e\ l \wedge \mathsf{eval}\ e'\ v \wedge \mathsf{mem}\ l\ v\ t \\
A4 & \forall l, v', t, \mathsf{mem}\ l\ v\ t \Rightarrow \mathsf{mem}\ l\ v'\ t \Rightarrow v = v' \\
A5 & \forall e, v, v', \mathsf{eval}\ e\ v \wedge \mathsf{eval}\ e\ v' \Rightarrow v = v' \\
A6 & \forall x, v, y, \mathsf{eval}\ x\ v \wedge x > y \Rightarrow v > y \\
A7 & \forall i, e, v, t, \mathsf{read}\ i\ e\ v\ t \Rightarrow \exists l, \mathsf{eval}\ e\ l \wedge \mathsf{lock}\ i\ l\ t \\
A8 & \forall i, x, t, \mathsf{lock}\ i\ x\ t \Rightarrow \mathsf{noYield}\ i\ x@(t, t'] \Rightarrow \mathsf{lock}\ i\ x\ t' \\
A9 & \forall i, x, e, t, y, j, t', \mathsf{lock}\ i\ x\ t \wedge \mathsf{yieldTo}\ i\ e\ y\ t' \\
 & \quad \wedge \mathsf{eval}\ e\ x \wedge \mathsf{eval}\ y\ j \wedge \mathsf{noYield}\ i\ x@(t, t') \\
 & \quad \Rightarrow \mathsf{lock}\ i\ x\ j\ t'
\end{array}$$

B. Substitution for contexts

$$\begin{array}{ll}
\mathcal{RT}[\![\cdot]\!] & = \cdot \\
\mathcal{RT}[\![\Theta, X{:}\mathsf{Type}]\!] & = \{\theta[X \mapsto \mathsf{C}] \mid \theta \in \mathcal{RT}[\![\Theta]\!] \text{ and } \mathsf{C} \in \mathsf{Type}\}
\end{array}$$

$$\begin{array}{ll}
\mathcal{RG}[\![\cdot]\!]^u_{\theta;\mathcal{T}} & = \{(k; \emptyset)\} \\
\mathcal{RG}[\![\Gamma, x{:}\tau]\!]^u_{\theta;\mathcal{T}} & = \{(k; \gamma[x \mapsto e]) \mid (k; \gamma) \in \mathcal{RG}[\![\Gamma]\!]^u_{\theta;\mathcal{T}} \\
 & \qquad \text{and } (k, e) \in \mathcal{RE}[\![\tau\gamma]\!]^u_{\theta;\mathcal{T}}\}
\end{array}$$