System M: A Program Logic for Code Sandboxing and Identification

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

Shayak Sen  
CS  
Carnegie Mellon University  
shayaks@cs.cmu.edu

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

Anupam Datta  
CS & ECE  
Carnegie Mellon University  
danupam@cmu.edu

Abstract

Security-sensitive applications that execute untrusted code often check the code’s integrity by comparing its syntax to a known good value or sandbox the code to contain its effects. System M is a new program logic for reasoning about such security-sensitive applications. System M extends Hoare Type Theory (HTT) to trace safety properties and, additionally, contains two new reasoning principles. First, its type system internalizes logical equality, facilitating reasoning about applications that check code integrity. Second, a confinement rule assigns an effect type to a computation based solely on knowledge of the computation’s sandbox. We prove the soundness of System M relative to a step-indexed trace-based semantic model. We illustrate both new reasoning principles of System M by verifying the main integrity property of the design of Memoir, a previously proposed trusted computing system for ensuring state continuity of isolated security-sensitive applications.

1. Introduction

Software systems, such as Web browsers, smartphone platforms, and extensible operating systems and hypervisors, are designed to provide subtle security properties in the presence of adversaries who can supply code, which is then executed with the privileges of the trusted system. For example, webpages routinely execute third-party JavaScript with full access to their content; smartphones execute apps from open app stores, often with very lax sandboxes; operating system kernels include untrusted (and often buggy) device drivers; and trusted computing platforms load programs from disk and only later verify loaded programs using the Trusted Platform Module (TPM) [32]. Despite executing potentially adversarial code, all these systems have security-related goals, often safety properties over traces [18]. For example, a hypervisor must ensure that an untrusted guest operating system running on top of it cannot modify the hypervisor’s page table, a webpage must ensure that an embedded untrusted advertisement cannot access a user’s password, and trusted computing mechanisms must enable a remote party to check that an expected software stack was loaded in the expected order on an untrusted server.

Secure execution of untrusted code in trusted contexts rely on two common mechanisms. First, 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. Second, code identification mechanisms are used to infer that an untrusted piece of code is in fact syntactically equal to a known piece of code. These mechanisms include distribution of signed code, and trusted computing mechanisms [32] that leverage hardware support to enable remote parties to check the identity of code on an untrusted computer. 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 sandboxing and code identification.

System M’s design is inspired by Hoare Type Theory (HTT) [21–23]. Like HTT, a monad separates computations with side-effects from pure expressions, and a monadic type both specifies the return type of a computation and includes a postcondition that specifies the computation’s side-effects. The postcondition of a computation type in System M uses predicates over the entire trace of the computation. This is motivated by our desire to verify safety properties [18], which are, by definition, predicates on traces. Further, the postcondition contains not one but two 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. The invariant assertion is directly used to represent safety properties.

To this basic infrastructure, we add two novel reasoning principles that internalize the rationale behind commonly used mechanisms for ensuring secure execution of adversary-supplied code: code identification and sandboxing. These rules derive effects of untyped code potentially provided by an adversary and, hence, enable the typing derivation of the trusted code to include as subderivations, the reasoning of effects of the adversarial code.

The first principle, a rule called EQ, ascribes the type of a program to another program $e'$: if $e$ is syntactically equal to $e'$ and $e : \tau$, then $e' : \tau$. 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 [25] 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 in 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).

Our second reasoning principle, manifest in a rule called CONFINE, allows us to type partially specified adversary-supplied code from knowledge of the sandbox in which the code will execute. The intuition behind this rule is that if all side-effecting interfaces available to a computation maintain a certain invariant on the shared state, then that computation cannot violate that invariant, irrespective of its actual code. The CONFINE rule generalizes prior work of Garg et al. on reasoning about interface-confined adversarial code in a first-order language [14]. The main difference from Garg et al. can be Booleans (\(\phi\) is function instantiation, or a suspended computation. Constants accommodate code, in addition to data, from the adversary and other trusted components. Our use of the CONFINE rule stresses our view that assumptions made about adversarial code should be minimized. In contrast, a lot of work, e.g., proof-carrying code [25], requires that...

We summarize System M’s term syntax in Figure 1. Pure expressions, we use call-by-name \(\beta\)-reduction, and the inclusion of adversary-supplied code make the model nonstandard.

System M is the first program logic that allows proofs of safety for programs that execute adversary-supplied code with adequate precautions, but does not force the adversarial code to be completely available for typing. Other frameworks like Bhargavan et al.’s contextual theorems [4] for F7 achieve expressiveness similar to the CONFINE rule for a slightly limited selection of trace properties. (We compare to related work in Section 7). Our step-indexed model of Hoare types is also novel; although our exclusion of preconditions, our use of call-by-name \(\beta\)-reduction, and the inclusion of 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 verifying the state continuity property of the design of Memoir [23], a previously proposed trusted computing system.

For reasons of space, we elide proofs, some technical details and accompanying technical appendix.

### 2. Term Language and Operational Semantics

We summarize System M’s term syntax in Figure 1. Pure expressions, denoted \(e\), are distinguished from effectful computations, denoted \(c\). An expression can be a variable, a constant, a function, a polymorphic function, a function application, a polymorphic function instantiation, or a suspended computation. Constants can be Booleans (\(\text{tt}, \text{ff}\)), natural numbers (\(n \in \mathbb{N}\)), thread identifiers (\(i \in \mathbb{Z}\)), and memory locations (\(l \in \mathbb{L}\)). We use \(\cdot\) as the place holder for the type in a polymorphic function instantiation. Suspended computations \(\text{comp}(c)\) constitute a monad with return \(\text{return}(e)\) and bind \(\text{letc}(1, x, c)\).

System M is parametrized over a set of action symbols \(A\), which are instantiated with concrete actions based on specific application domains. For instance, \(A\) may be instantiated with memory operations such as read and write. An action, denoted \(a\), is the application of an action symbol \(A\) to expression arguments.

---

**Figure 1. Term Syntax**

A basic computation is either an atomic action (\(\text{act}(a)\)) or \(\text{return}(e)\) that returns the pure expression \(e\) immediately, \(\text{fix}(f)(x).c\) is a fixpoint operator \(f\), which represents a suspended fixpoint computation, may appear free in the body \(c\). Computation \(c\) is the application of a fixpoint computation to its argument. \(\text{letc}(1, x, c)\) denotes the sequential composition of \(c_1\) and \(c_2\), while \(\text{letc}(1, x, c)\) 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 \(x \leftarrow e_1; c_2\) and \(\text{let} x = e_1; c_2\). When the expression returned by the first computation is not used \(c_2\), we write \(e_1; c_2\) and \(c_1; c_2\).

The operational semantics of System M are step-small and based on interleaving of concurrent threads.

**Stack**

\[ K ::= \epsilon | x.c ::= K \]

**Thread**

\[ T ::= \langle i; K; e \rangle | \langle i; K; e \rangle | \langle i; \text{stuck} \rangle \]

**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 \(\langle i; K; e \rangle\) or \(\langle i; K; e \rangle\), where \(i\) is a unique identifier of that thread (drawn from a set \(I\) of such identifiers), \(K\) is the execution (continuation) stack, and \(e\) and \(e\) are the computation and expression currently being evaluated. A thread permanently enters a stuck state, denoted \(\langle i; \text{stuck} \rangle\), 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 \(x.e\), \(x\) binds the return expression of the computation preceding \(c\). A configuration of the system is a shared state \(\sigma\) and a set of all threads. \(\sigma\) is application-specific; for the rest of this paper, we assume that it is a standard heap mapping pointers to expressions, but this choice is not essential. For example, in modeling network protocols, the heap could be replaced by the set of undelivered (pending) messages on the network.

For pure expressions, we use call-by-name \(\beta\)-reduction \(\rightarrow\). This choice simplifies the operational semantics and the soundness proofs, as explained in Sections 4. We elide the standard rules for \(\rightarrow\). The small-step transitions for threads and system configurations are shown in Figure 2. The relation \(\sigma \triangleright T \rightarrow \sigma' \triangleright T'\) defines a small-step transition of a single thread. \(\sigma \triangleright C\) denotes a small-step transition for configuration \(C\); it results from the reduction of any single thread in \(C\).

The rules for \(\sigma \triangleright T \rightarrow \sigma' \triangleright T'\) are mostly straightforward. The rules for evaluating an atomic action (R-ActS and R-ActTF) rely on a function next that takes the current store \(\sigma\) 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 next \(\sigma, a\) returns \((\sigma’, \text{stuck})\).

If the action returns a non-stuck expression \(e\) (rule R-ActS), then the top frame \(x.e\) is popped off the stack, and \(e; x.e\) becomes the current computation of the thread. If next returns stuck (rule R-ActTF), then the thread enters the stuck state and permanently remains there. When a sequencing statement \(\text{letc}(1, x, c)\) is evaluated, the frame \(x.c_2\) is pushed onto the stack, and \(e_1\) is first reduced to a suspended computation \(\text{comp}(c_1)\); then \(c_1\) is evaluated.
We briefly review Memoir \[28\], our main application, and highlight its state snapshot. 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. The service would want to ensure secrecy and integrity of its state; in this case, the set of stored passwords. Simply encrypting and signing the service’s state cannot prevent attacks. For the password manager service, this attack could cause the service to respond with old (possibly compromised) passwords. Memoir solves this problem by using the TPM to provide state integrity guarantees. Memoir relies on the following TPM features:

- **Platform configuration registers (PCRs)** contain 20-byte hashes known as measurements that summarize the current configuration of the system. The value they contain can only be updated in two ways: (1) a reset operation which sets the value of the PCR to a fixed default value; (2) an extend operation which takes as argument a value \( v \) and updates the value of the PCR to the hash of the concatenation of its current value with \( v \).

- **Late launch** is a command that can be used to securely load a program. It extends the hash of the textual reification of the program into a special PCR (PCR17). Combined with the guarantees provided by a PCR, late launch provides a mechanism for precise code identification.

- **Non-volatile RAM (NVRAM)** provides persistent storage that allows access control based on PCR measurements. Specifically, permissions on NVRAM locations can be tied to a PCR \( p \) and value \( v \) such that the location can only be read when the value contained in \( p \) is \( v \).

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 to 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, a piece of service code, and a snapshot. In Figure 3 we show a snippet of the Memoir service invocation code. Memoir retrieves the key and freshness tag from the NVRAM. Memoir then 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, 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 to disk.

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 of security for Memoir requires reasoning about the effects the service, which is provided by potentially malicious parties.

To derive properties of the runmodule code shown above one needs to assign a type to srvc, which is provided by an adversary. The service srvc, run on line 6, is a function that contains no free actions. However, srvc takes as arguments interface functions corresponding to every atomic action in our model. Shown above are ExtendPCR and ResetPCR which are simply wrappers for the corresponding atomic actions.

For example, the proof requires deriving the following two invariant properties about srvc:

1. It does not change the value of the PCR to a state that allows the adversary to later read the NVRAM.
2. It does not leak the secret key.

The first invariant is derived using the fact that the service is confined to the interface exposed by the TPM. The second invariant is derived in three steps: (i) prove that \( \text{src} \) is syntactically equal to the initial service; (ii) assume that the initial service does not leak the secret key; and (iii) hence infer that \( \text{src} \) does not leak the secret key. We next describe System M’s typing rules that enable such reasoning.

### 3.2 Typing Adversary Supplied Code

#### Reasoning about effects of confinement

In analyzing 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. In our Memoir example, every program on the machine is confined to the interface provided by the TPM. Using just this confinement information, we can sometimes deduce a useful effect-type for the partially trusted program. Suppose \( c \) is a closed computation, which syntactically does not contain any actions and can invoke as subprocedures the computations \( c_1, \ldots, c_n \) only (i.e., \( c \) is confined to \( c_1, \ldots, c_n \)). If all actions performed by \( c_1, \ldots, c_n \) satisfy a predicate \( \varphi \), then the actions performed by \( c \) must also satisfy \( \varphi \), irrespective of the code of \( c \). Hence, we can statically specify the effects of \( c \), without knowing its code, but knowing the effects of \( c_1, \ldots, c_n \).

We formalize this intuition in a typing rule called \( \text{CONFIN} \). To explain this rule, we introduce some notation. Let \( \tau \) denote types in System M that include postconditions for computations and, specifically, let \( \text{cmap}(\tau, \varphi) \) denote the monadic type of computations that return a value of type \( \tau \) and whose actions satisfy the predicate \( \varphi \). (The notation \( \text{cmap}(\tau, \varphi) \) is simpler than our actual computation types, but it suffices for the explanation here.)

As an illustration of our \( \text{CONFIN} \) rule, consider any closed expression \( e \). Assume that \( e \) does not contain any primitive actions. Then, we claim that for any \( \varphi \), \( e \) has the type \( \text{cmap}(\text{bool}, \varphi) \rightarrow \text{cmap}(\text{bool}, \varphi) \). To understand this claim, assume that \( \varphi \) is the property “the action is not a write to memory”. To show that \( e : \text{cmap}(\text{bool}, \varphi) \rightarrow \text{cmap}(\text{bool}, \varphi) \), we must show that for any \( v : \text{cmap}(\text{bool}, \varphi) \), \( e \) evaluates to \( v \). Here, we must show that the actions performed by the computation, say \( c \), that \( e \) evaluates to do not include write. This can be argued easily: Because \( e \) is closed and does not contain any actions, the only way this computation \( e \) can write is by invoking the computation \( v \). However, because \( v : \text{cmap}(\text{bool}, \varphi) \), \( v \) does not write. Hence, \( e : \text{cmap}(\text{bool}, \varphi) \).

In fact, we can assign \( e \) any type, including higher-order function types, as long as the effects in that type are \( \varphi \). Let the predicate \( \text{conf}e(\tau)(\varphi) \) mean that \( \varphi \) \( \equiv \varphi \) for all nested types of the form \( \text{comp}(\tau', \varphi') \) in \( \tau \). Let \( \text{conf}e:\left(\Gamma\right)(\varphi) \) mean that every type \( \tau \) that \( \Gamma \) maps to satisfies \( \text{conf}e(\tau)(\varphi) \). Let \( \text{fa}(e) = \emptyset \) mean that \( e \) syntactically does not contain any actions. Then, the idea of typing through confinement is captured by the following rule. The rule says that for any \( e \) without any actions, if \( \tau \)'s nested effects are \( \varphi \), and the types of the free variables in \( e \) also have \( \varphi \) as effects, then \( e : \tau \) with any predicate \( \varphi \). (Our actual typing rule, shown in Section 4 after more notation has been introduced, is more complex. The actual rule also admits predicates over traces, which are more general than predicates over individual actions that we have considered here.)

\[
\frac{\text{fa}(e) = \emptyset \quad \text{conf}e:\left(\tau\right)(\varphi) \quad \text{for } (e) \in \Gamma \quad \text{conf}e:\left(\Gamma\right)(\varphi)}{\Gamma \vdash e : \tau} \quad \text{CONFIN}
\]

In our Memoir example, we use the \( \text{CONFIN} \) rule to derive the invariants of the service invoked by the attacker. For instance, if we can show that each of the TPM primitives do not reset the value of the PCR, then using the \( \text{CONFIN} \) rule, we can claim that \( \text{src} \), when applied to these primitives does not reset the value of the PCR. We revisit this proof with specific details in Section 4.2.

In typing a statically unknown expression using the \( \text{CONFIN} \) rule we assume that the expression is syntactically free of actions and that all of its free variables are in \( \Gamma \). These are reasonable assumptions for untrusted code to be sandboxed. In an implementation these assumptions can be discharged either by dynamic checks during execution, by static checks during program linking, or by hardware-enforced interface confinement. For example, in our Memoir analysis, the hardware ensures that TPM state can be modified by the service only using the TPM interface.

#### Deriving properties based on code integrity

Next we need to show that \( \text{src} \) does not leak its secret key. We assume this property about the initial service Memoir was invoked with. (This property could be verified either by manual audits or automated static analysis of the service code). However, in our model the adversary could invoke Memoir on malicious service code (e.g., replacing a legitimate password manager service with code of the adversary’s choice). In this case, we can show with additional reasoning that \( \text{src} \) invoked later must be the same program as the initial service.

To allow typing \( \text{src} \), based on the proof of equality with the initial service and an assumed type for the initial service, we add a new rule called Eq.

\[
\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash e = e' : \text{true}}{\Gamma \vdash e' : \tau} \quad \text{Eq}
\]

The Eq rule assigns the type \( \tau \) of any expression \( e \) to any other expression \( e' \), which is known to be syntactically equal to \( e \). This rule is trivially sound.

This pattern of first establishing code identity (identify an unknown code with some known code) and then using it to refine types is quite common in proofs of security-relevant properties. A similar pattern arises in analysis of systems that rely on memory protections to ensure that code read from the shared memory is the same as a piece of trusted code, and therefore, safe to execute. In Datta et al.’s work on analysis of remote attestation protocols [11], similar patterns arise for typing potentially modified software executed in a machine’s boot sequence. Their model is untyped, but if it were to be typed, Eq could be used to complete the proofs.

### 4. Type System and Assertion Logic

The syntax for System M types is shown in Figure 4. Types for expressions, denoted \( \tau \), include type variables \( (X) \), a base type \( \text{bool} \), dependent function types \( (\Pi \times \tau_1, \tau_2) \), and polymorphic function types \( (\forall X. \tau) \). Since System M focuses on deriving trace properties of programs, the difference between base types such as \( \text{unit} \) and \( \text{bool} \) is of little significance. Therefore, System M has one base
type b to classify all first-order terms. The type any contains all syntactically well-formed expressions (any stands for “untyped”). Memory always stores expressions of type any because the adversary could potentially write to any memory location.

Similar to HTT, a suspended computation comp(c) is assigned a monadic type comp(η), where η is a closed computation type. A closed computation type u1.u2.i.(x.τ φ1, φ2) contains two post-conditions, φ1 and φ2. Both are interpreted relative to a trace T, φ1, the partial correctness assertion, holds whenever a computation of this type finishes execution on the trace. It is parameterized by the id i of the thread that runs the computation, the interval [u0, u1] during which the computation runs and the return value x of the computation. φ2, called the invariant assertion, holds while a computation of the computation type is still executing (or is stuck), but has not returned. It is parameterized by the id i of the thread running the computation and the time interval [u0, u1] over which the computation has executed. Formally, a suspended computation comp(c) has type comp(u1.u2.i.(x.τ φ1, φ2)) if the following two properties hold for every trace T: (1) if a thread i on trace T begins to run c at time U1 and at or before U2, c returns an expression e, then e has type τ, and T satisfies φ2[U1, U2, i, e/u1, u2, i, x][τ], (2) if a thread i on trace T begins to run c at time U1 and at time U2, c has not finished, then T satisfies φ2[U1, U2, i/u1, u2, i, τ]. The meaning of all types is made precise in Section 5.2.

The type η may be either a partial correctness assertion, an invariant assertion, or a pair of both. Fixpoint computations have the type Π x:τ. u1.u2.i.(y:τ, φ1, φ2), discussed in more detail with typing rules. If Φ has this type, then for any e : τ, (f e) is a recursive computation of closed computation type u1.u2.i.(y:τ, φ1, φ2)[f/e].

Assumptions, denoted ϕ, are standard first-order logical formulas interpreted over traces. Atomic assertions are denoted P.

We write α to categorize actions. A fully applied action has the type Act(η), where η denotes the action’s effects.

4.1 Typing Rules

Our typing judgments use several contexts. Θ is a list of type variables. The signature Σ contains specifications for action symbols. Π contains logical variable type bindings. These variables can only be of the type b or any. Γ contains dependent variable type bindings. Δ contains logical assertions. The ordered context Ξ = u0, u1, i provides reference time points and a thread id to typing judgments for computations. When typing a computation, [u0, u1] are parameters representing the interval during which the computation executes and i is a parameter representing the id of the thread that executes the computation. A summary of the typing judgments is shown below.

| Θ; Σ; Γ; Δ ⊢ e : τ | expression e has type τ | fixed-point computation c has type η | computation c has type η | φ holds while reductions are non-effectful | φ is true |
|----------------------|------------------------|----------------------------------|------------------------|----------------------------------|------------|
| u:b; Θ; Σ; Γ; Δ ⊢ c : τ | fixed-point computation c has type η | computation c has type η | φ holds while reductions are non-effectful | φ is true | Θ; Σ; Γ; Δ ⊢ φ silent |

When typing expressions and fixpoint computations, u is earliest time point when the term can be evaluated on the trace. The first three judgments are indexed by a qualifier Q, which can either be empty or u0, u1, i, φ, which we call an invariant. Variables u0, u1, and i have the same meaning as the context Ξ, and may appear free in φ. Rules indexed with u0, u1, i, φ are used for deriving properties of programs that execute adversarial code. Roughly speaking, the context Γ in these rules contains variables that are place holders for expressions that satisfy the invariant φ. We explain here some selected rules of our type system; the remaining rules are listed in the accompanying technical appendix.

Silent threads

Reductions on a trace can be categorized into those induced by the rules R-ActS and R-ActF in Figure 3, and those induced by other rules. We call the former effectful and the latter non-effectful or silent. The typing judgment Ξ; Θ; Σ; Γ; Δ ⊢ φ silent specifies properties of threads while they perform only silent reductions or do not reduce at all. The judgment is auxiliary in proofs of both partial correctness and invariant assertions, as will become clear soon. The following rule states that if φ is true, then a trace containing a thread’s silent computation satisfies φ.

| Ξ; Θ; Σ; Γ; Δ ⊢ φ true | Ξ; Θ; Σ; Γ; Δ ⊢ φ ok | Ξ; Θ; Σ; Γ; Δ ⊢ φ silent |

The type system may be extended with other sound rules for this judgment. For instance, the following is a trivially sound rule: u0, u1, i; Θ; Σ; Γ; Δ ⊢ c : x.τ,φ means that if in trace T any thread with id i begins to execute computation c at time U1, and at time U2, c returns an expression e, and T satisfies the formulas in Δ, then e has type τ, and T also satisfies φ2[U1, U2, i/e/u1, u2, i, x].

In rule Act, the type of an atomic action is directly derived from the specification of the action symbol in a. We elide rules for the judgment a := Act(u1, u2, i.(x.τ φ1, φ2)), which derives types for actions based on the specifications in Σ. We explain the invariant assertions for actions with the discussion of invariant typing for computations. When typing a, the logical variable typing context includes u2 : b and i : b, because they may appear free in τ and Δ.

For brevity, we elide the types for variables of type b, as they are obvious from the context.

Rule Ret assigns e’s type to ret(c). The trace T containing the evaluation of ret(e) satisfies two properties, which appear in the postcondition of ret(c). First, the return expression, which is bound to x, is e (assumption (x = e)). Second, T satisfies any property φ that φ silent holds. This is because reduction of ret(e) is silent. Here e is typed under the time point u0, indicating that e can only be evaluated after u2.

Rule SeqC types the sequential composition letc(c1, c2). Starting at time point u0 and returning at u3, the execution of letc(c1, x, c2) in any thread i can be divided into three segments for some u1, u2: between time u0 and u1, where thread i takes only a silent step, pushing x, c2 onto the stack; between time u1 and u2, where the computation c1 runs; and between time u2 and u3, where c2 runs. The first three premises of SeqC assert the effects of each of these three segments. When type checking c2, the facts learned from the execution so far (φ0 and φ1) are included in the context. The fourth premise checks that φ is the logical consequence of the conjunction of the three evaluation segments’ properties.

The above rules have the same qualifier Q in the premises and the conclusion. Rule SeqCCOMP combines derivations with different qualifiers in a sequencing statement. The Γ context in the typing of c1 and c2 must be empty. Because the free variables in c1 are place holders for expressions that satisfy an invariant φ1, while the free variables in c2 are for ones that satisfy a different invariant φ2, c1 and c2 cannot share free variables except those in Γ. Note that both Q and Q2 can be empty. This rule is necessary for typing the sequential composition of two programs that contain differently sandboxed code: c1 executes sandboxed code that satisfies φ1, and c2 either contains no sandboxed programs, or ones that satisfy φ2.
Partial correctness typing

\[
\begin{align*}
&u_1; \Theta; \Sigma; \Gamma', u_2; i; \Gamma; \Delta \vdash Q \ a \ :: \ \text{act}(u_0, u_e, i, (x: x_1, \varphi_2)) \\
u_1, u_2; i; \Theta; \Sigma; \Gamma'; \Gamma; \Delta \vdash \varphi \ \text{silent} \\
\forall \theta \in \text{dom}(\Gamma) & \quad \text{let } \gamma = (u_1, u_2, i / u_0, u_e, i) \\
\frac{d; \Theta; \Sigma; \Gamma'; \Gamma; \Delta \vdash u_1, u_2, i. (x: x_1, \varphi_2) \land \varphi \ \text{ok}}{u_1, u_2, i; \Theta; \Sigma; \Gamma'; \Gamma; \Delta \vdash \text{act}(x: x_1, \varphi_2) \land \varphi}
\end{align*}
\]

Invariant typing

\[
\begin{align*}
&\Theta; \Sigma; \Gamma'; u_0, u_3, i; \Gamma; \Delta \vdash \varphi \ \text{ok} \\
u_0, u_1, i; \Theta; \Sigma; \Gamma'; u_3; \Gamma; \Delta, u_0 \leq u_1 \vdash \varphi_0 \ \text{silent} \\
u_0, u_3, i; \Theta; \Sigma; \Gamma'; u_0 : b, u_3; \Gamma; \Delta, u_0 \leq u_2 \vdash \varphi_0 \ \text{silent} \\
\frac{c_1 : x_1, \varphi_1}{u_2, u_3, i; \Theta; \Sigma; \Gamma'; u_0, u_1; x : \tau; \Delta, u_2 \leq u_3, \varphi_0, \varphi_1 \ \text{ok}} \\
\frac{c_2 : y, \varphi_2}{u_2, u_3, i; \Theta; \Sigma; \Gamma'; u_0, u_1; y : \tau; \Delta \vdash (\varphi_0 \land \varphi_2)} \\
\Theta; \Sigma; \Gamma'; u_2, u_3, i; \Gamma, \Delta \vdash \varphi_1 \ \text{ok} \\
\frac{u_0, u_3, i; \Theta; \Sigma; \Gamma'; \Gamma; \Delta \vdash Q \ letc(c_1, x, c_2) : y: \tau \ \varphi}{\text{SEQCOMP}}
\end{align*}
\]

Figure 5. Selected Rules for Computation Typing

Invariant typing for computations

The meaning of the invariant typing judgment \( u_1, u_2, i; \Theta; \Sigma; \Gamma'; \Gamma; \Delta \vdash \varphi \) is the following: Assuming that on a trace \( T \), thread \( t \) begins to execute \( c \) at time \( U_1 \), and at time \( U_2 \) \( c \) has not yet returned (this includes the possibility that \( c \) is looping indefinitely or is stuck), if \( T \) satisfies assumptions in \( \Delta \), then \( T \) also satisfies \( \varphi[U_1, U_2 / i / u_0, u_e, i] \).

We first explain the invariant assertions for actions (rule ACT). The thread executing the atomic action is silent before the action returns. Therefore, the invariant assertion of the action is the conjunction of the invariant specified in \( \Sigma \) and the effect of being silent.

Next, we explain the rule SEQCI for the sequencing statement \( \text{letc}(c_1, x, c_2) \). We need to consider three cases when deriving the invariant assertion \( \varphi \) of \( \text{letc}(c_1, x, c_2) \) in the interval \( [u_0, u_2] \): (1) the computation has not started until \( u_2 \) (the computation \( c_2 \) has not yet returned), (2) the computation of \( c_2 \) has not yet returned until \( u_3 \), and (3) the computation of \( c_3 \) has returned, but \( c_2 \) has not returned until \( u_3 \). The first five premises of rule SEQCI establish properties of a silent thread, the partial correctness and invariant assertions of the computation in \( c_1 \), and the invariant assertion of \( c_2 \). The next three judgments check that in each of the three cases (1)–(3), the final assertion \( \varphi \) holds.

For example, \( \varphi[\text{comp}(u_0, u_e, i, r: \forall \lambda \text{letc}(c_1, x, c_2)) \vdash Q \varphi \) can be assigned the following type. Predicate \( (\lambda \text{letc}(c_1, x, c_2)) \) is true when at time \( u \), memory location \( l \) is allocated and stores the expression \( e \). Predicative \( e \varphi \) is true if \( \beta \)-reduces to \( e' \), which cannot reduce further. Write \( l \varphi e \) states that thread \( l \varphi \) writes to address \( e \) at time \( u \). The partial correctness assertion states that this suspended computation returns what’s stored in the location that \( e \) reduces to. The invariant assertion states that during its execution, the thread executing it does not write to the memory: \( \varphi[\text{comp}(u_0, u_e, i, r: \forall \lambda \text{letc}(c_1, x, c_2)) \vdash Q \varphi \).
in Section 3.1 is that now $\varphi$ is a predicate over an interval and a thread in a trace, not just a predicate over individual actions. The intuitive idea behind the rule is similar: If $e$ is a computation that is free of actions and confined to use the computations $c_1, \ldots, c_n$ for interaction with the shared state, and each of the computations $c_1, \ldots, c_n$ maintain a trace invariant $\varphi$ while they execute, then as $e$ executes, it maintains $\varphi$.

Technically, because $\varphi$ also accepts as arguments any intervals on a trace (it has free variables $u_1, u_2$), we require that $\varphi$ be trace composable, meaning that if $e$ holds on two consecutive intervals of a trace, then it holds across the union of the intervals. Formally, $\varphi$ is trace composable if $\forall u_1, u_2, u_3, i, t$. ($\varphi(u_1, u_2, i) \land \varphi(u_2, u_3, i)$) $\Rightarrow$ $\varphi(u_1, u_3, i)$. Further $\varphi$ has to hold on intervals when thread $i$ is silent. This prevents us from deriving arbitrary properties of untrusted code. For instance, $\varphi$ cannot be $\bot$. (No trace can satisfy the invariant $\bot$.) This rule relies on checking that $\tau$ relates to the invariant $\varphi$, represented as the relation $\text{confine} \ (\tau \; \{u_0, u_2, i, \varphi\}$. This relation means that $\varphi$ is both the partial correctness assertion and the invariant assertion in every computation $\text{comp}(e)$ occurring in $\tau$. Similarly, $\Gamma$ is required to map every free variable in $e$ to a type that satisfies the same relation. The conclusion is indexed by the invariant $u_0, u_2, i, \varphi$ to record the fact that all substitutions for variables in $\Gamma$ need to satisfy $\varphi$.

\begin{equation}
\text{confine} \ (\tau \; \{u_0, u_2, i, \varphi\}
\end{equation}

\begin{equation}
\text{confine} \ (\tau_1 \; \{u_0, u_2, i, \varphi\} \quad \text{confine} \ (\tau_2 \; \{u_0, u_2, i, \varphi\}
\end{equation}

\begin{equation}
\text{confine} \ (\Pi \tau, \tau_g \; \{u_0, u_2, i, \varphi\}
\end{equation}

\begin{equation}
\text{confine} \ (\tau \; \{u_0, u_2, i, \varphi\}
\end{equation}

\begin{equation}
\text{confine} \ (\text{comp}(u_0, u_2, i, \varphi)) \; \{u_0, u_2, i, \varphi\}
\end{equation}

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

The rule CONF-SUB constrains a regular typing derivation to a specific invariant $u_0, u_2, i, \varphi$. This is sound because the first premise does not require the substitutions for $\Gamma$ to satisfy any specific invariant, so they can be narrowed down to any invariant. The conclusion must be tagged with the invariant $\varphi$, because: (1) $\tau$ could be a base type, in which case, the invariant is not evident in $e$’s type; and (2) the types in $\Gamma$ are allowed to contain nested effects that are not $\varphi$. Reason (1) is also why the conclusion of the CONFINE rule is indexed.

Finally, the time point enables expression types to include facts that are established by programs executed earlier. For example, the return type of $\text{letc(a1; z, ret\text{(comp(a2))})}$ can be the following, assuming that the effect of action $a_1$ is $A_1 \; i$, and $a_2$ is $A_2 \; i$, $\text{comp}(u_b, u_e, i, (r, b \; \exists \; u, u_b < u \leq u_e \land A_2 \; i \; u \land \exists j, u', u'< u \land A_1 \; j \; u', \top))$.

We wouldn’t have been able to know that $A_1$ happens before $A_2$ without the time point in the expression typing rules.

**Logical Reasoning** System M includes a proof system for first-order logic, most of which is standard. We show here the rule HONEST, which allows us to deduce properties of a thread based on the invariant assertion of the computation it executes.

\begin{equation}
\text{HONEST} 
\end{equation}

If we know that a thread $i$ starts executing at time $u$ with payload computation $c$ (premise $\text{start}(c, u)$) and computation $c$ has an invariant postcondition $\varphi$, then we can conclude that at any later point $u'$, $\varphi$ holds for the interval $(u, u')$. The condition that $c$ be typed under an empty $\Gamma$ context is required by the soundness proofs, which we discuss in Section 5.4.

4.2 Examples

We prove the following state continuity property of Memoir. It states that after the service has been initialized at time $u_1$ with the key $\text{sk}

\begin{equation}
\forall u_i, \text{state}_i, \text{skey}_i, \text{init}_i, \text{sinit}_i \text{service}_i(\text{init}_i, \text{sk}, \text{service}, \text{sinit}_i) \Rightarrow \forall u > u_i, \text{service}_i \text{try}_i(\text{sk}, \text{state}_i) \Rightarrow \text{state}_i \text{service}_i(\text{init}_i, \text{sk}, \text{state}_i) \Rightarrow \exists j, u' < u, ((\exists \text{service}_i \text{invoke}_i(j, \text{sk}, s, \text{state}_i) \Rightarrow \text{service}_i \text{try}_i(j, \text{sk}, \text{state}_i) \Rightarrow \text{service}_i \text{init}_i(j, \text{sk}, \text{state}_i) \Rightarrow \text{service}_i(\text{init}_i, \text{sk}, \text{state}_i) \Rightarrow \text{service}_i(\text{init}_i, \text{sk}, \text{state}_i)) \Rightarrow (u', u))
\end{equation}

The expressiveness of the first-order logic enables us to specify the above property, where the ordering of events is crucial. For the full proofs, we refer the reader to our technical appendix. We now revisit our discussion in Section 3 and highlight critical uses of the System M program logic in the proof. Recall that Memoir has two phases: service initialization and service invocation. During initialization, we assume that the Memoir module runmodule (Figure 3) is assigned NVRAM location Nloc and service service. The permission for accessing Nloc (which stores the secret key used to encrypt state and the freshness tag) is set to the value of PCR 17. This PCR stores a nested hash $h_{\text{hash}} = H(h || \text{code_hash(service)})$. Here, the term $H(x)$ denotes hash of $x$. || denotes concatenation, $h$ is any value and $\text{code_hash(x)}$ is a hash of the textual reification of program $x$. After initialization, we prove the following two key invariants about executions of runmodule:

1. **PCR Protection**: The value of PCR 17 contains the value $h_{\text{hash}}$ only during late launch sessions running runmodule.
2. Key Secrecy: If the key corresponding to a service is available to a thread, then it must have either generated it or read it from Nloc.

We prove these invariants using the HONEST rule, which requires us to type runmodule. Since runmodule invokes srcv, we need to type srcv. Recall that srcv is adversarially-supplied code. Thus, in typing it we make use of the CONFINE and EQ rules.

For the first invariant, we derive the necessary type for srcv by typing against the TPM interface. The particular invariant type we wish to derive about srcv is that in a late launch session if the value in the PCR has been set to a value that is not a prefix of $s_{hash}$, then srcv cannot change the value in the PCR to something that is a prefix of $s_{hash}$ (i.e., it cannot fool the NVRAM access control mechanism into believing that service was loaded when it was not).

$\text{srcv \ ExtendPCR \ ResetPCR \ \cdots \ (state, \ req) :}$

$\Downarrow \text{cmp}(u_b, u_e, i, \neg\text{PCRPrefix}(pcr17, s_{hash}) \oplus u_b) \Rightarrow \forall u \in (u_b, u_e]. \ (\text{InLLSession}(u, \text{runmodule}, \tau) \Rightarrow \neg\text{PCRPrefix}(pcr17, s_{hash}) \oplus u_b)$

To derive this type using the CONFINE rule, it is sufficient to show that each function in the TPM interface can be assigned this type. For example, the $\text{ExtendPCR}$ interface satisfies this invariant as it can only extend a PCR value. This derivation is a key step in proving that the service does not change the value of the PCR to a state that allows any entity other than $\text{runmodule}$ to read the NVRAM location $Nloc$ (i.e., the first invariant of srcv in Section 3.1).

Similarly, we can prove that the permissions on Nloc are always tied to PCR 17 being $s_{hash}$, by typing srcv with the invariant that the permissions on Nloc cannot be changed. Thus, whenever Nloc is read from, the value of PCR 17 is $s_{hash}$. We also show separately that in any particular instance of runmodule with srcv, the state of PCR 17 must be $H(h)[\text{code}_{hash}(srcv)]$ for some h. Therefore, by Nloc’s access control mechanism, we prove that $H(h)[\text{code}_{hash}(srcv)] = s_{hash}$ and therefore srcv = service (where = denotes syntactic equality).

This is a key step to proving the key secrecy invariant. It allows us to transfer assumptions about the known Memoir service service to the adversarially-supplied service srcv. Specifically, we assume that srcv has the following type $\tau_{src}$ (which means that if the input of service does not contain a secret s then the output doesn’t contain it) and an invariant KeepsSecret(i, s, Nloc) (which means that s is not sent out on the network and the only NVRAM location s possibly written to is Nloc).

$\tau_{src} = \Pi \cdot \text{msg}\cdot \text{cmp}(u_b, u_e, i, \forall x. \forall s. \neg\text{Contains}(i, s) \Rightarrow \neg\text{Contains}(x, s), \forall s. \neg\text{Contains}(i, s) \Rightarrow \text{KeepsSecret}(i, s, Nloc) \cdot (u_b, u_e, i))$.

Using the above assumption about service and the proof that srcv = service, we use EQ to derive the required type for srcv (i.e., the second invariant of srcv discussed in Section 3.1).

5. Semantics and Soundness

We build a step-indexed semantic model [3] for types and prove soundness of System M relative to that. Central to the semantics is the notion of invariant. We build two sets of semantics: one is a semanticsx for invariants of the form $u_b, u_e, i, \varphi$ ($\mathcal{R}_\text{INV}^{\{u_b, u_e, i, \varphi\}}$), and the other is an invariant-indexed semantics for types ($\mathcal{R}_\text{INV}^{\{u_b, u_e, i, \varphi\}}[	au]$). These two sets coincide when confine ($\tau$) ($u_b, u_e, i, \varphi$) holds (Lemma 1).

5.1 A Step-indexed Semantics for Invariants

We define $\mathcal{R}_\text{INV}^{\Phi}[	au]_{u_b, u_e, i, \varphi}$, $\mathcal{R}_\text{INV}^{\Phi}[	au]_{u_b, u_e, i}$, and $\mathcal{R}_\text{INV}^{\Phi}[	au]_{u_b, u_e}$ ($\Phi = u_b, u_e, i, \varphi$), the sets of step-indexed normal forms, expressions, and computations that satisfy the invariant $\varphi$ respectively. $\tau$ is the trace that the term is evaluated on and $u$ is the earliest time point when the term is evaluated. These sets categorize invariant-confined adversarial programs.

We first define the set of step-indexed computations that satisfy an invariant $\varphi$ below. An indexed computation $(k, c)$ belongs to this relation if the following holds: (1) during any interval $u_b$ and $u$ when thread $i$ executes $c$ on $T$, $\varphi[u_b, u_e, i/u_b, u_e, i]$ holds on $T$ and (2) if $c$ completes at time $u_e$, then the expression that $c$ returns, indexed by the remaining steps of the trace, satisfies the same invariant.

$\mathcal{R}_\text{INV}^{\{u_b, u_e, i, \varphi\}}[	au]_{u_b} = \{(k, c) \mid \forall u_b, u_e, i, u \leq u_b \leq u_e,\}$

where $\gamma = [u_b, u_e, i/u_b, u_e, i]$.

$\mathcal{R}_\text{INV}^{\{u_b, u_e, i, \varphi\}}[	au]_{u_b} = \{(k, c) \mid \forall u_b, u_e, i, u \leq u_b \leq u_e, \}$

where $\gamma = [u_b, u_e, i/u_b, u_e, i]$.

$\mathcal{R}_\text{INV}^{\{u_b, u_e, i, \varphi\}}[	au]_{u_b} = \{(k, c) \mid \forall u_b, u_e, i, u \leq u_b \leq u_e, \}$

where $\gamma = [u_b, u_e, i/u_b, u_e, i]$.

$\mathcal{R}_\text{INV}^{\{u_b, u_e, i, \varphi\}}[	au]_{u_b} = \{(k, c) \mid \forall u_b, u_e, i, u \leq u_b \leq u_e, \}$

where $\gamma = [u_b, u_e, i/u_b, u_e, i]$.

$\mathcal{R}_\text{INV}^{\{u_b, u_e, i, \varphi\}}[	au]_{u_b} = \{(k, c) \mid \forall u_b, u_e, i, u \leq u_b \leq u_e, \}$

where $\gamma = [u_b, u_e, i/u_b, u_e, i]$.

$\mathcal{R}_\text{INV}^{\{u_b, u_e, i, \varphi\}}[	au]_{u_b} = \{(k, c) \mid \forall u_b, u_e, i, u \leq u_b \leq u_e, \}$

where $\gamma = [u_b, u_e, i/u_b, u_e, i]$.

$\mathcal{R}_\text{INV}^{\{u_b, u_e, i, \varphi\}}[	au]_{u_b} = \{(k, c) \mid \forall u_b, u_e, i, u \leq u_b \leq u_e, \}$

where $\gamma = [u_b, u_e, i/u_b, u_e, i]$.

The definition of $\mathcal{R}_\text{INV}^{\{u_b, u_e, i, \varphi\}}[	au]_{u_b}$, relation is standard: if $c$ evaluates to a normal form $n$ in $m$ steps, then $n$ has to be in the value relation indexed by the number of the remaining steps.
Types Interpreting an expression type \( \tau \) as 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 \( \eta \), that cannot be reduced further under call-by-name \( \beta \)-reduction. \( C \) contains the set of all possible indices and all syntactically well-formed normal forms. This is used to interpret the type \( \tau \) of any untyped programs. As usual, we require that \( C \) be closed under reduction of step-indices. Let \( P(S) \) denote the powerset of \( S \). The set of all semantic types is denoted \( Type \).

Type definition \( Type \) defines the value and expression interpretations of expression types \( \tau \) (written \( \text{RV}(\Phi)[P] \)) and \( \text{RE}(\Phi) \), as well as the interpretation of computation types \( \eta \) (written \( \text{RC}(\Phi)[\eta] \)) simultaneously by induction on types \( \Phi \). Let \( \theta \) denote a partial map from type variables to Type, \( T \) denote the trace that expressions are evaluated on, and \( u \) denote the time point after which expressions are evaluated.

The interpretation of the base type \( b \) is the same as \( \text{RV}(\Phi)[b] \). The type \( b \) itself doesn’t specify any effects, and, therefore, expressions in the interpretation of \( b \) only need to satisfy the invariant \( \Phi \). The interpretation of the function type \( II \) is nonstandard: the substitution for the variable \( x \) is an expression, not a value. This simplifies the proof of soundness of function application: since System M uses call-by-name \( \beta \)-reduction, the reduction of \( e_1 \) need not evaluate \( e_2 \) to a value before it is applied 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 \( u' \), which is later than \( u \), the function application belongs to the interpretation of the argument type with the index \( j \) and time point \( u' \). The interpretation of the function type also includes normal forms that are not \( \lambda \) abstractions that are in the \( \text{RE}(\Phi) \) relation. These are adversary-supplied untyped code, which is required by our type system to satisfy the invariant \( u_0, u_i, \eta \).

The interpretation of the monadic type includes suspended computations \( (k, \text{comp}(c)) \) such that \( (k, c) \) belongs to the interpretation of computation types, defined below. Because \( c \) executes after \( u \), the beginning and ending time points selected for evaluating \( c \) are no earlier than \( u \). Similar to the interpretation of the function type, the interpretation of the monadic type also includes normal forms that are not monads, but satisfy the invariant \( u_0, u_i, \eta \). The interpretation of the any type \( \tau \) contains all normal forms.

We lift the value interpretation \( \text{RV}(\Phi)[P] \) to the expression interpretation \( \text{RE}(\Phi) \) in a standard way.

Interpretation of formulas Formulas are interpreted on traces. We write \( T \models \varphi \) to mean that \( \varphi \) is true on trace \( T \).

We assume a valuation function \( v(\tau) \) that returns the set of atomic formulas that are true on the trace \( T \). For first-order quantification, we select terms in the denotation of the type \( (\tau) \), which is defined as follows:

- \( \text{any} \) \( \in \{ e \mid e \) is an expression \( \}
- \( b \) \( \in \{ e \mid e \to bv \}
- \( \Pi \langle x.e, e, e \rangle \in \{ e \mid e_1 \) is an element of \( \tau \}

The types of the logical variables can only be \( b \), any and function types. The interpretation of these types is much simpler than that of expressions.

Interpretation of computation types The interpretation of a computation type, \( \text{RC}(\Phi)[\eta] \), is a set of step-indexed computations. The trace \( T \) contains the execution of the computation. \( \Xi = u_0, u_i, i \) has its usual meaning, except that \( u_0, u_i, \) and \( i \) are concrete values, not variables.

We define the semantics of the partial correctness types, denoted \( \text{RC}(\Phi)[\eta] \) below. Informally, it contains the set of indexed computations \( c \) if \( T \) contains a complete execution of the computation \( c \) in the time interval \( (u_0, u_i) \) in thread \( i \) such that \( c \) returns \( e' \) at time \( u_0 \) and it is also the case that \( T \) satisfies \( v(\varphi) \) and \( e' \) has type \( \tau \) semantically. Similar to the \( \text{RE}(\Phi) \) relation, these remaining steps include not just steps of the thread executing \( c \), but also other threads. The invariant \( u_0, u_i, \varphi \) is used in the specification of the return value.

We illustrate some key points of our semantic model. We instantiate the next function (Section 2) for the read action as follows:

\[
\text{next}(\sigma, \text{read } \xi, \eta) = \begin{cases} 
(\sigma, \sigma(\xi)) & \text{if } \xi \in \text{dom}(\sigma) \\
(\sigma, \text{stick}) & \text{if } \xi \notin \text{dom}(\sigma)
\end{cases}
\]

Predicate stuck \( u \) is true when thread \( i \) is in the stuck state at time \( u \). The first example below shows the semantic specification of the read action. The partial correctness assertion states that as long as the location \( l \) being read is allocated when the read happens, the thread does not get stuck and the expression \( y \) returned by \( \text{read} \) is the in-memory content \( v \) of the location read. The invariant assertion states that between the time the read action becomes the redex and the time it reduces, the thread is not stuck.

1. \( (\text{n, act(\text{read } e))} \in \text{RC}(\Phi)[y; \text{any}, \forall l, \text{mem } l \to u_2 \land \text{eval } e l \to (y = e) \land \text{~stuck } v \) \)
2. \( \text{RC}(\Phi)[\forall l, i, \text{write } \text{Write } l e l t] \subseteq \)
thread performs a write action at any time, is the empty set. This is
because the semantics of invariant assertions require that any
trace containing the execution of such a computation satisfy this
invariant. A trivial counterexample is a trace containing a second
thread that writes to memory.

5.4 Soundness of the Logic

We prove that our type system is sound relative to the semantic
model of Section 4. We start by defining valid substitutions
for contexts. We write $\mathcal{R}(s)$ to denote the set of valid
substitutional assignments of $s$. We write $\mathcal{R}(s)[\emptyset] = \{ s \in \mathcal{R}(s) \}$ to denote a set
of assignments for variables in $s$. Each indexed substitution is a pair
of an index and a substitution $\gamma$ for variables.

We prove two key lemmas. Lemma 1 states that when all the
effects in $\tau$ are $u_b, u_e, i, \varphi$, then the interpretation of $\tau$
is the same as the interpretation of the invariant $u_b, u_e, i, \varphi$. The proof is
by induction on the structure of $\tau$.

**Lemma 1.** (Index types are confined). $\tau :: (\pi_1 \rightarrow \pi_2)(\pi_1 \rightarrow \pi_2)$ implies $\mathcal{R}(\pi_1 \rightarrow \pi_2)[\emptyset] = \mathcal{R}(\pi_1 \rightarrow \pi_2)[\emptyset] = \emptyset$.

The following lemma states that if $e$ does not contain any actions,
then $e$, with its free variables substituted by expressions
that satisfy an invariant $u_b, u_e, i, \varphi$, satisfies the same invariant.
The proof is by induction on the structure of $\tau$.

**Lemma 2.** (Invariant confinement). If $\varphi$ is composable, and $\varphi$ silent between time $u_b$ and $u_e$, and $e :: \tau$ implies $\mathcal{R}(\tau)[\emptyset] = \emptyset$, then $\mathcal{R}(\tau)[\emptyset] = \emptyset$.

The soundness theorem (Theorem 3) has two different statements
for judgements with the empty qualifier and the invariant qualifier.
The ones for judgments with an empty qualifier state that
for any invariant $\varphi$, if the substitution for $\tau$ belongs to the
interpretation of types, then the expression (computation) belongs to
the interpretation of its type, indexed by the same invariant $\varphi$.

For judgments qualified by a specific invariant $\varphi$, the soundness
theorem statements are also specific to that $\varphi$.

**Theorem 3 (Soundness).**

Assume that $\forall A :: \alpha \in \Sigma, \forall \varphi, \tau, u, n, (n, A) \in \mathcal{R}(A)[\emptyset] = \tau$.

1. (a) $\mathcal{E} :: u :: \mathcal{E} \cap \Sigma \cap \Gamma; \Delta \vdash e :: \varphi, \tau \in \mathcal{R}(s)$ implies $\mathcal{R}(\mathcal{E})[\emptyset] = \emptyset$.

2. (b) $\mathcal{E} :: u, u_b, i, \varphi, \tau \vdash c :: \varphi, \tau \in \mathcal{R}(s)$. If $\mathcal{E} \cap \Sigma \cap \Gamma$ is a $\Delta$-cocompatible $\varphi$ then $\mathcal{E} \cap \Sigma \cap \Gamma$ is $\tau$-cocompatible $\varphi$.

(1) $\mathcal{E} :: u :: \mathcal{E} \cap \Sigma \cap \Gamma; \Delta \vdash e :: \varphi, \tau \in \mathcal{R}(s)$ implies $\mathcal{R}(\mathcal{E})[\emptyset] = \emptyset$.

(2) $\mathcal{E} :: u, u_b, i, \varphi, \tau \vdash c :: \varphi, \tau \in \mathcal{R}(s)$ implies $\mathcal{R}(\mathcal{E})[\emptyset] = \emptyset$.

We prove the soundness theorem by induction on typing derivations
and a subinduction on step-indices for the case of fixpoints.

The proof of soundness of the rule CONFINE (2.2(a)) first uses
Lemma 1 to show that a substitution $\gamma$ for $\Gamma$ where $\gamma$ maps each
variable in $\Gamma$ to the type interpretation of $\Gamma(x)$ is also a substitution
where $\gamma(x)$ belongs to the interpretation of the invariant. Then we
use Lemma 2 to show that the untyped term $e \gamma$ belongs to the
interpretation of the invariant. Applying Lemma 1 again, we can show
that $e \gamma$ is in the interpretation of $\tau$. The confine relations in the
premises are key to this proof. The proof of the rule CONF-SUB
uses the induction hypothesis directly: a derivation with an empty
qualifier can pick substitutions with any invariant $\varphi$.

To prove the soundness of HONEST, we need to show that given
any substitution $(n, \gamma)$ for $\Gamma$, the trace satisfies the invariant of $c$.
From the last 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 $(m, \gamma)$ for $\Gamma$.
Because $\Gamma$ is empty, we complete the proof by using the induction
hypothesis on the first premise given an empty substitution $(m, \gamma)$.

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, including those that are adversarial.

The theorem holds because we account for adversarial actions in
the definition of $\mathcal{R}(u_b, u_e, i, \varphi)[\emptyset]$ for $\emptyset$. A similar theorem holds
for partial correctness assertions.

**Theorem 4 (Robust safety).** If

- $u_1, u_2, i, \Delta \vdash c :: \varphi, \tau \in \Delta$.
- $\mathcal{E}$ is a trace obtained by executing the parallel composition of
  threads of ID $(t_1, \ldots, t_k)$.
- At time $U_b$, the computation that thread $t_j$ is about to run is $c$
- At time $U_e$, $c$ has not returned
then $\mathcal{T} \models \varphi[U_b, U_c, t_3/u_1, u_2, v]$.

6. Discussion

Proving non-stickness We can use System M’s invariant assertions to verify that a program always remains non-stuck. Recall the example from Section 5. We can prove non-stuckness for a computation $e$ by showing that it has the invariant postcondition $(\neg \text{stuck}) o (u, t, u_e)$. To complete such a proof, we would require that all action types assert non-stuckness in their postconditions under appropriate assumptions on the past trace. For instance, the first example in Section 5.4 states that we can assert non-stuckness in the postcondition of the read action, if the memory location being read has been allocated.

Choice of reduction strategy System M uses call-by-name $\beta$-reduction for expressions, which simplifies the operational semantics as well as the soundness proofs. Other evaluation strategies we have considered force us to use $\beta$-equality in place of syntactic equality in Eq. This makes the system design, semantics, and soundness proofs very complicated. In particular, the Eq rule that uses $\beta$-equality cannot be proven sound in a model where expressions are indexed by their reduction steps.

7. Related Work

Hoare Type Theory (HTT) In HTT [21–23], a monad classifies effectful computations, and is indexed by the return type, a precondition 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 Confine). We also exclude standard Hoare pre-conditions from computation types. Usually, pre-conditions ensure that well-typed programs do not get stuck. We argued in Section 6 that in System M this property can be established for individual programs using only invariant postconditions. The standard realizability semantics of HTT [29] are based on a model of CPOs, whereas our model is syntactic and step-indexed [30].

RHTT [24] is a relational extension of HTT used to reason about access 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 with capabilities of both. 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 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 [14]. The main conceptual difference from LS is that in System M trusted and untrusted components may exchange code and data, whereas in LS this interface is limited to data. Our Confine 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. The difference is that System M’s rule allows typing higher-order expressions, which makes it more complex, e.g., we must index the typing derivations with invariants and define interpretations for invariants based on step-indexing programs to obtain soundness. LS itself is based on a logic for reasoning about Trusted Computing Platforms [10] and Protocol Composition Logic (PCL) for reasoning about safety properties of cryptographic protocols [9].

Rely-guarantee reasoning There are two broad kinds of techniques to prove invariants over state shared by concurrent programs. Coarse-grained reasoning followed in, e.g., Concurrent Separation Logic (CSL) [6] and the concurrent version of HTT [25], assumes clearly marked critical regions and allows programs to violate invariants on shared state only within them. This assumes that resource contention is properly synchronized, which is generally unrealistic when executing concurrently with an unspecified adversary. In contrast, fine-grained reasoning followed in, e.g., the method of Owicki-Gries [26] and its successor, rely-guarantee reasoning [17], makes no synchronization assumption, but has a higher proof burden at each individual step of a computation. In proofs with System M, including the Memoir example in this paper, we use a template for rely-guarantee reasoning taken from LS. The methods used to prove invariants within this template are different because of the new higher-order setting.

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 [1]. Gordon and Jeffrey develop a very widely used proof technique for proving robust safety based on this type [15]. 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 $\mathcal{RC}(\Phi)[u]_{\phi,T,X}$ in Section 5.2). Due to this generalized semantic definition, robust safety (Theorem 4) is again a trivial consequence of soundness (Theorem 3).

Several type systems for establishing different kinds of safety properties build directly or indirectly on the work of Abadi [1] and Gordon and Jeffrey [15]. Of these, the most recent and advanced are RCF [3] and its extensions [4, 31]. 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 [13]. 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 [10]. 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.

$F^*$ [11] extends F with quantified types, a rich kinding system, concrete refinements and several other features taken from the language Fine [30]. 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 Confine rules that statically derive computational effects of untyped adversary-supplied code.
Code-Carrying Authorization (CCA) [20] is another extension to [15] 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 construct 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 CONFINE rule assigns types statically.

**Verification of TPM and Protocols based on TPM**

Existing work on verification of TPM APIs and protocols relying on TPM APIs uses a variety of techniques [19, 21]. Gurgens et al. uses automaton to model the transitions of TPM APIs [12]. Several results [11, 13, 14] used the automated tool Proverif [5]. Proverif translates protocols encoded in Pi calculus into horn clauses. To check security properties such as secrecy and correspondence, the tool runs a resolution engine on these horn clauses and clauses representing an Dolev-Yao attacker. Proverif over-approximates the protocols such as ones that use TPM PCRs [12]. 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 [12], as it requires reasoning about higher-order code.

A proof of safety formalized in TLA+ [15] was presented in the Memoir paper [28]. They showed that Memoir’s design refines an obviously safe specification that cannot be rolled back thus implying the state integrity property we prove. However, this proof assumes that the service being protected is a constant action with no effects. Consequently, they do not need to reason about the service program being changed or causing unsafe effects. Our proofs assume a more realistic model requiring that the identity of the service be proven and that the effects of the service be analyzed based on the sandbox provided by the TPM.

8. Conclusion

System M is a program logic for proving safety properties of programs that may execute adversary-supplied code with some precautions. System M generalizes Hoare Type Theory with invariant assertions, and adds two novel typing rules—EQ and CONFINE—that allow typing adversarial code using reasoning in the assertion logic and assumptions about the code’s sandbox, respectively. We prove soundness and robust safety relative to a step-indexed, trace model of computations. Going further, we would like to build tools for proof verification and automatic deduction in System M.

References

[1] M. Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5), 1999.
[2] A. Ahmed. Step-indexed syntactic relations for recursive and quantified types. In *Proc. ESOP*, 2006.
[3] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. *TOPLAS*, 33(2):8:1–8:45, 2011.
[4] K. Bhargavan, C. Fournet, and A. D. Gordon. Modular verification of security protocol code by typing. In *Proc. POPL*, 2010.
[5] B. Blanchet. Using Horn clauses for analyzing security protocols. In V. Cortier and S. Kremer, editors, *Formal Models and Techniques for Analyzing Security Protocols*, volume 5 of *Cryptography and Information Security Series*, pages 86–112. IOS Press, Mar. 2011.
[6] S. Brookes. A semantics for concurrent separation logic. *Theoretical Computer Science*, 375(1-3):227–270, 2007.
[7] L. Chen and M. Ryan. Attack, solution and verification for shared authorisation data in TCG TPM. In *Proc. FAST’09*, 2010.
[8] A. Datta, A. Derek, J. C. Mitchell, and D. Pavlovic. A derivation system and compositional logic for security protocols. *Journal of Computer Security*, 13(3):423–482, 2005.
[9] A. Datta, A. Derek, J. C. Mitchell, and A. Roy. Protocol Composition Logic (PCL). *Electronic Notes in Theoretical Computer Science*, 172: 311–358, 2007. ISSN 1571-0661.
[10] 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.
[11] S. Deaune, S. Kremer, M. D. Ryan, and G. Steel. A formal analysis of authentication in the TPM. In *Proc. FAST’10*, 2011.
[12] S. Deaune, S. Kremer, M. D. Ryan, and G. Steel. Formal analysis of protocols based on TPM state registers. In *Proc. CSF’11*, 2011.
[13] C. Fournet, A. D. Gordon, and S. Maffeis. A type discipline for authorization policies. *TOPLAS*, 29(5), 2007.
[14] 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*, 265:49–71, 2010.
[15] A. D. Gordon and A. Jeffrey. Authenticity by typing for security protocols. *Journal of Computer Security*, 11(4):451–519, July 2003.
[16] S. Gurgens, 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.
[17] C. B. Jones. Tentative steps toward a development method for interferring programs. *TOPLAS*, 5(4):596–619, 1983.
[18] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977.
[19] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. ISBN 032114306X.
[20] S. Maffeis, M. Abadi, C. Fournet, and A. D. Gordon. Code-carrying authorization. In *Proc. ESORICS’08*, 2008.
[21] A. Nanevski, A. Ahmed, G. Morrisett, and L. Birkedal. Abstract predicates and mutable ADTs in Hoare type theory. In *Proc. ESOP’07*, 2007.
[22] A. Nanevski, G. Morrisett, and L. Birkedal. Hoare type theory, polymorphism and separation. *Journal of Functional Programming*, 18 (5&6):865–911, 2008.
[23] A. Nanevski, P. Goverceau, and G. Morrisett. Towards type-theoretic semantics for transactional concurrency. In *Proc. TLDI’09*, 2009.
[24] A. Nanevski, A. Banerjee, and D. Garg. Verification of information flow and access control policies via dependent types. In *Proc. IEEE S&P*, 2011.
[25] G. C. Necula. Proof-carrying code. In *Proc. POPL*, 1997.
[26] S. Owicki and G. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976.
[27] B. Parno, J. M. McCune, and A. Perrig. Bootstrapping trust in commodity computers. In *Proc. S&P*, pages 414–429, 2010.
[28] 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.
[29] R. L. Petersen, L. Birkedal, A. Nanevski, and G. Morrisett. A realizability model for impredicative Hoare type theory. In *Proc. ESOP’08*, 2008.
[30] N. Swamy, J. Chen, and R. Chugh. Enforcing stateful authorization and information flow policies in fine. In *Proc. ESOP*, 2010.
A. Semantics

Semantics for invariant properties

Next we define a logical relation indexed only by an invariant property $u_b, u_c, i, \varphi$.

$$\mathbb{R}_V \varsubsetneq \{ (k, n) | n \neq \lambda x.e, \Lambda \times e, \text{comp}(c) \}$$

The configuration at time $b$ and $e$ in the configuration at time $k$, $c$, $x.c'::K; c \cdots$ between $u_b$ and $u_c$, the stack of thread $i$ always contains prefix $x,c':K$.

$$\Rightarrow T \vdash \varphi \cap$$

$ \{ (k, c) | \forall u_b, u_c, i, \varphi \} = \{ (k, c) | \forall u_b, u_c, i, \varphi \} \\
(\gamma, e') \in \mathbb{R}_V \varsubsetneq \{ (k, n) | n \neq \lambda x.e, \Lambda \times e, \text{comp}(c) \} \\
\Rightarrow (j, e'[x]) \in \mathbb{R}_V \{ (k, n) | n \neq \lambda x.e, \Lambda \times e, \text{comp}(c) \}$

Semantics for invariant indexed types

Figure 2 summarizes the interpretation of types indexed by the invariant property $u_b, u_c, i, \varphi$.

The invariant property is used to constrain the behavior of expressions that evaluate to normal forms that do not agree with their types.

$$\mathbb{R}(u_b, u_c, i, \varphi) \varsubsetneq \{ (k, c) | j_b \geq j_c \}$$

The configuration at time $u_b, u_c, i, \varphi$ between $u_b$ and $u_c$, the stack of thread $i$ always contains prefix $x,c':K$.

$$\Rightarrow T \vdash \varphi \cap$$

$ \{ (k, c) | \forall u_b, u_c, i, \varphi \} = \{ (k, c) | \forall u_b, u_c, i, \varphi \} \\
(\gamma, e') \in \mathbb{R}(u_b, u_c, i, \varphi) \varsubsetneq \{ (k, n) | n \neq \lambda x.e, \Lambda \times e, \text{comp}(c) \} \\
\Rightarrow (j, e'[x]) \in \mathbb{R}(u_b, u_c, i, \varphi)$

B. Term Language and Operational Semantics

Syntax

Base values

| $b$ | $v$ | $i$ | $\ell$ | $n$ |
|-----|-----|-----|-------|-----|

Expressions

| $e$ | $x.b$ | $x.e$ | $\Lambda.X.e$ |
|-----|-------|-------|------------|

Actions

| $a$ | $\alpha$ | $\cdot$ |
|-----|----------|-------|

Computation

$$\text{act}(a) \cdot \text{ret}(c) \cdot \text{fix}(f(x.c)).c \cdot e$$

| $c_1$ | $c_2$ | $c_1 \cdot c_2$ |
|-----|-----|----------|

if $e$ then $c_1$ else $c_2$

Beta reductions

We define the $\beta$-reduction rules below.
Figure 8. Semantics for inv-indexed types

\[
e \rightarrow_β e'
\]

\[
(\lambda x.e_2) \rightarrow_β e[e_2/x]
\]

\[
\lambda X.e \rightarrow_β e
\]

\[
e_1 \rightarrow_β e'_1
\]

\[
e_1 e_2 \rightarrow_β e'_1 e_2
\]

\[\Rightarrow \begin{array}{c}
\Theta \vdash \Sigma \text{ ok} \\
\Theta \vdash \Sigma \text{ ok} \\
\Theta \vdash \Sigma, A :: \alpha \text{ ok}
\end{array}
\]

\[\Rightarrow \begin{array}{c}
\Theta \vdash \Sigma, A :: \alpha \text{ ok} \\
\Theta \vdash \Sigma, \Delta \text{ ok}
\end{array}
\]

\[
\sigma \vdash T \Rightarrow \sigma' \Rightarrow T'
\]

\[
\text{next}(\sigma, a) = (\sigma', e) \quad e \neq \text{ stuck}
\]

\[
\sigma \vdash (i; x.e :: K; \text{act}(a)) \Rightarrow \sigma' \vdash (i; K; e[e/x])
\]

\[\text{R-ACTS}
\]

\[
\text{next}(\sigma, a) = (\sigma', \text{stuck})
\]

\[
\sigma \vdash (i; x.e :: K; \text{act}(a)) \Rightarrow \sigma' \vdash (i; \text{stuck})
\]

\[\text{R-ACTF}
\]

\[
\sigma \vdash (i; \text{stuck}) \Rightarrow \sigma \vdash (i; \text{stuck})
\]

\[\text{R-STUCK}
\]

\[
\sigma \vdash (i; x.e :: K; \text{ret}(e)) \Rightarrow \sigma \vdash (i; K; e[e/x])
\]

\[\text{R-RET}
\]

\[
\sigma \vdash (i; K; \text{le}t(e_1, x.e_2)) \Rightarrow \sigma \vdash (i; x.e_2 :: K; e_1)
\]

\[\text{R-SEQE1}
\]

\[
\sigma \vdash (i; K; e \rightarrow_β e') \Rightarrow \sigma \vdash (i; K; e')
\]

\[\text{R-SEQE2}
\]

\[
\sigma \vdash (i; x.e_2 :: K; \text{comp}(c_1)) \Rightarrow \sigma \vdash (i; x.e_2 :: K; e_1)
\]

\[\text{R-SEQE3}
\]

\[
\sigma \vdash (i; K; \text{letc}(c_1, x.e_2)) \Rightarrow \sigma \vdash (i; x.e_2 :: K; c_1)
\]

\[\text{R-SEQC}
\]

\[
\sigma \vdash (i; K; \text{fix}(f)(x).e) \Rightarrow \sigma \vdash (i; K; c[\lambda z. \text{comp}(f)(x)(z)]/f)[e/x])
\]

\[\text{R-FIX}
\]

\[
C \Rightarrow C'
\]

\[
\sigma \vdash T \Rightarrow \sigma' \Rightarrow T'
\]

\[
\sigma \vdash T, T_1, \ldots, T_n \Rightarrow \sigma' \Rightarrow T', T_1, \ldots, T_n
\]

C. Well-formedness Judgments

Well-formedness judgments for contexts and types

\[
fv(e_1) \cup fv(e_2) \subseteq \text{dom}(\Gamma)
\]

\[\Gamma \vdash e_1 \equiv e_2 \text{ ok}
\]
D. Typing Rules

Typing for simple terms  \[ \Gamma \vdash e : \tau \]

- \[ x : \tau \in \Gamma \]
- \[ \Gamma, x : \tau_1 \vdash e : \tau_2 \]
- \[ \Gamma \vdash \Pi x : \tau_1. \tau_2 \]

Confine relation

\[ \text{confine} (b) \ (u_b, u_e, i, \varphi) \]

\[ \text{confine} (\tau_1) \ (u_b, u_e, i, \varphi) \]

\[ \text{confine} (\tau_1, \tau_2) \ (u_b, u_e, i, \varphi) \]

Typing rules for expressions

\[ \text{confine} \ (\varphi) \ (u_b, u_e, i, \varphi) \]

Typing rules for silent threads

- \[ \Xi, \Theta; \Sigma; \Gamma^\ell; \Delta \vdash \varphi \text{ silent} \]

Typing rules for actions
Logical reasoning rules

\[ \Theta; \Sigma; \Gamma, u : b, \Delta \vdash \varphi \] HONEST

\[ u_1, u_2, i; \Theta; \Sigma; \Gamma, \Delta \vdash c : \varphi \] HONEST

\[ \Theta; \Sigma; \Gamma, \Delta \vdash \forall u'.b. (u > u) \Rightarrow \varphi[u, u', \Gamma / u_1, u_2, i] \] HONEST

Typing rules for computations We summarize the typing rules for computations in Figures 9 and 10.

E. Semantics

Semantics for invariant properties Next we define a logical indexed only by an invariant property \( u_B, u_E, i, \varphi \).

\[ \mathcal{R}_{INV}[u_B, u_E, i, \varphi] = \{ (k, c) \mid \langle k, e \rangle \in \mathcal{R}_{INV}[u_B, u_E, i, \varphi] \} \]

and

\[ \langle k, e \rangle \in \mathcal{R}_{INV}[u_B, u_E, i, \varphi] \iff (j, e) \in \mathcal{R}_{INV}[u_B, u_E, i, \varphi] \]

The configuration at time \( u_B \) is \( u_B \rightarrow \sigma_B \cdot \cdots \cdot \langle i; x, c' : K; c \rangle \cdots \) between \( u_B \) and \( u_E \) (inclusive), the stack of thread \( i \) always contains prefix \( x, c' : K \)

\[ \mathcal{T} \models \varphi \]

\[ \langle k, c \rangle \mid \forall k, u_B, u_E, i, u \leq u_B \leq u_E, \text{let } \gamma = [u_B, u_E, i / u_1, u_2, i] \]

The configuration at time \( u_B \) is \( u_B \rightarrow \sigma_B \cdot \cdots \cdot \langle i; x, c' : K; c \rangle \cdots \) between \( u_B \) and \( u_E \), the stack of thread \( i \) always contains \( x, c' : K \)

\[ \langle j, e' \rangle \in \mathcal{R}_{INV}[u_B, u_E, i, \varphi] \]
Fixpoint
\[ u : b ; \Theta ; \Sigma ; \Gamma^I ; \Delta ; \Gamma \vdash Q : \eta \]
\[ \Gamma_1 = y : \tau , f : \Pi \gamma . \tau . \text{comp}(u_1 , u_3 , i . (x : \tau_1 , \phi , \phi')) \]
\[ u_1 : b ; u_2 : b , i : b ; \Theta ; \Sigma ; \Gamma^I ; \Delta , u \leq u_1 \leq u_2 \vdash \phi_0 \text{ silent} \]
\[ u_2 , u_3 , i : \Theta ; \Sigma ; \Gamma^I , u_1 : b ; u : b ; \Theta ; \Gamma_1 , \Delta , u_2 < u_3 \vdash \phi_0 \vdash \phi : c \vdash \tau_1 , \phi_1 \]
\[ u_2 , u_3 , i : \Theta ; \Sigma ; \Gamma^I , u_1 : b ; u : b ; \Theta ; \Gamma_1 , \Delta , u_2 \leq u_3 \vdash \phi_0 \vdash \phi : \varphi_2 \]
\[ \Theta ; \Sigma ; \Gamma^I , u_1 : b ; u : b ; u_2 : u_3 : b , i : b ; \Gamma_1 , x : \tau_1 , \Delta \vdash (\phi_0 \land \varphi_1) \Rightarrow \phi \text{ true} \]
\[ \Theta ; \Sigma ; \Gamma^I , u_1 : b ; u : b ; u_2 : u_3 : b , i : b ; \Gamma_1 , \gamma : \tau , \Delta \vdash (\phi_0 \land \varphi_2) \Rightarrow \phi \text{ true} \]
\[ \Theta ; \Sigma ; \Gamma^I , u_1 : b ; u_3 : b , i : b ; u : b ; \Gamma , y : \tau_1 , \Delta \vdash \phi_0 / u_3 / u_2 \vdash \phi' \text{ true} \]
\[ \Theta ; \Sigma ; \Gamma^I , u_1 : b ; u_3 : b , i : b ; u : b ; \Gamma , y : \tau_1 , \Delta \vdash \phi_0 \land \varphi_1 \land \varphi_2 \Rightarrow \phi \text{ true} \]
\[ \Theta ; \Sigma ; \Gamma^I , u_1 : b ; u_3 : b , i : b ; u : b ; \Gamma , y : \tau_1 , \Delta \vdash (\phi_0 \land \varphi_1 \land \varphi_2) \Rightarrow \phi \text{ true} \]
\[ \Theta ; \Sigma ; \Gamma^I , u_1 : b ; u_3 : b , i : b ; u : b ; \Gamma , y : \tau_1 , \Delta \vdash \phi_0 \land \varphi_1 \land \varphi_2 \Rightarrow \phi \text{ true} \]

Partial correctness typing
\[ \exists ; \Theta ; \Sigma ; \Gamma^I ; \Delta \vdash Q : \Gamma \vdash \phi (y) \subseteq \text{dom}(\Gamma) \]
\[ \text{APP} \]
\[ u_1 : b ; u_2 : b , i : b ; \Theta ; \Sigma ; \Gamma^I , u_3 : b , i : b ; \Gamma , \Delta \vdash Q : \phi \vdash \phi (y) \subseteq \text{dom}(\Gamma) \]
\[ \text{ACT} \]
\[ u_1 : b ; u_2 : b , i : b ; \Theta ; \Sigma ; \Gamma^I , u_3 : b , i : b ; \Gamma , \Delta \vdash Q : \phi (a) : x : \tau_1 , \varphi_1 [u_1 , u_2 , i / u_3 , u_e , j , \varphi_2 [u_1 , u_2 , i / u_3 , u_e , j] \land \varphi) \]
\[ \text{SEQE} \]
\[ u_0 : b , u_3 : b , i : b ; \Theta ; \Sigma ; \Gamma^I , u_3 : b , i : b ; \Gamma , \Delta \vdash Q : \text{letc}(e_1 , e_2) : y : \tau_1 , \varphi \]
\[ \text{SEQC} \]
\[ u_2 : b ; \Theta ; \Sigma ; \Gamma^I , u_3 : b , i : b ; \Theta ; \Sigma ; \Gamma^I , \Delta \vdash Q : \text{letc}(e_1 , e_2) : y : \tau_1 , \varphi \]
\[ \text{RE1} \]
\[ u_1 : b , u_2 : b , i : b ; \Theta ; \Sigma ; \Gamma^I , \Delta \vdash Q : \text{letc}(e_1 , e_2) : x : \tau \leq \varphi \]

Figure 9. Computation typing rules (1)
Invariant typing

$$\Xi; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash Q c : \eta$$

$$\Theta; \Sigma; \Gamma^L, u_0 : b, u_3 : b, i : b; \Gamma; \Delta \vdash \varphi_0 \text{ ok}$$

$$u_0 : b, u_3 : b, i : b; \Theta; \Sigma; \Gamma^L, u_3 : b; \Gamma, \Delta, u_0 \leq u_1 \vdash \varphi_0 \text{ silent}$$

$$u_0 : b, u_3 : b, i : b; \Theta; \Sigma; \Gamma^L, \Gamma; \Delta, u_0 \leq u_3 \vdash \varphi_0 \text{ silent}$$

$$u_1 : b; \Theta; \Sigma; \Gamma^L, u_0 : b, u_3 : b, i : b; \Gamma, \Delta, u_0 \leq u_1 \vdash \varphi_0 \text{ silent}$$

$$u_2 : b; \Theta; \Sigma; \Gamma^L, u_0 : b, u_3 : b, i : b; \Gamma, \Delta, \tau, u_1 : b, x : \tau, u_1 < u_2 \leq u_3, \varphi_0[u_1, u_2, i/u, u_3, j] \vdash Q c_2 : \varphi_2$$

$$\Theta; \Sigma; \Gamma^L, u_0 : b, u_3 : b, i : b; \Gamma, \Delta \vdash \varphi_0 \Rightarrow \varphi \text{ true}$$

$$\Theta; \Sigma; \Gamma^L, u_0 : b, u_3 : b, i : b; \Gamma, u_1 : b, \Delta \vdash \varphi_0 \text{ silent}$$

$$u_0 : b, u_3 : b, i : b; \Theta; \Sigma; \Gamma^L; \Gamma, \Delta \vdash Q \text{ let(e1, x, c2)} : \varphi$$

$$\Theta; \Sigma; \Gamma^L, u_0 : b, u_3 : b, i : b; \Theta; \Sigma; \Gamma^L; \Gamma, \Delta \vdash \varphi \text{ silent}$$

$$u_0 : b, u_3 : b, i : b; \Theta; \Sigma; \Gamma^L; \Gamma, \Delta \vdash Q \text{ ret(e)} : \varphi$$

$$\Theta; \Sigma; \Gamma^L, u_0 : b, u_2 : b, i : b; \Theta; \Sigma; \Gamma^L; \Gamma, \Delta \vdash \text{ if e then c1 else c2} : \varphi$$

$$u_1 : b, u_2 : b, i : b; \Theta; \Sigma; \Gamma^L; \Gamma, \Delta \vdash Q c : \varphi_0 \text{ silent}$$

$$u_1 : b, u_2 : b, i : b; \Theta; \Sigma; \Gamma^L; \Gamma, \Delta, u_0 \leq u_1 \vdash \varphi_0 \text{ silent}$$

$$u_1 : b, u_2 : b, i : b; \Theta; \Sigma; \Gamma^L; \Gamma, \Delta, u_0 \vdash Q (\text{eval e tt}) : \varphi_1$$

$$\Theta; \Sigma; \Gamma^L, u_0 : b, u_2 : b, i : b; \Theta; \Sigma; \Gamma^L; \Gamma, \Delta \vdash \varphi_0 \Rightarrow \varphi$$

$$\Theta; \Sigma; \Gamma^L, u_0 : b, u_2 : b, i : b; \Theta; \Sigma; \Gamma^L; \Gamma, \Delta \vdash Q \text{ proj(e1, x, \varphi_2)} : \varphi_2$$

$$\Theta; \Sigma; \Gamma^L, u_0 : b, u_2 : b, i : b; \Theta; \Sigma; \Gamma^L; \Gamma, \Delta \vdash Q \text{ cutc} : \eta$$

$$\Xi; \Theta; \Sigma; \Gamma^L; \Theta; \Sigma; \Gamma^L; \Gamma, \Delta_1 \vdash \varphi \text{ true}$$

$$\Xi; \Theta; \Sigma; \Gamma^L; \Theta; \Sigma; \Gamma^L; \Gamma, \Delta_1, \Delta_2 \vdash Q c : \eta$$

Figure 10. Computation typing (2)
\[ u_0 : b, u_2 : b \vdash i : b; \Theta; \Sigma; \Gamma^L, u_3 : b; \Delta, u_0 \leq u_1 \leq u_2 \vdash \varphi_0 \silent \]
\[ u_1 : b; \Theta; \Sigma; \Gamma^L, u_0 : b, u_2 : b, u_3 : b; i : b; \varphi_0 \vdash c_1 : \text{comp}(u_b, u_e, j, (x, \tau, \varphi_1, \varphi_1')) \]

Let \( \gamma = [u_b, u_e, i/u_b, u_e, j] \)

\[ u_0, u_3 : b; i : b; \Theta; \Sigma; \Gamma^L, u_1 : b; \Delta, u_0 < u_2, \varphi_0, \varphi_1 \vdash \varphi_2 c_2 : y : \tau, \varphi \]

\( \Theta; \Sigma; \Gamma^L ; u_0 : b, u_3 : b; i : b; \Gamma, u_1 : u_2 b, y : \tau; \Delta \vdash (\varphi_0 \land \varphi_1 \land \varphi_2) \Rightarrow \varphi \true \)

\( \Theta; \Sigma; \Gamma^L; u_0 : b, u_3 : b; i : b; \Gamma, y : \tau \vdash \varphi \ok \)

\[ u_0 : b, u_3 : b; i : b; \Theta; \Sigma; \Gamma^L; \Delta \vdash \varphi (e_1 ; c_2) : y : \tau, \varphi \]

\( \Theta; \Sigma; \Gamma^L, u_0 : b, u_3 : b; i : b; \Theta; \Sigma; \Gamma^L, u_3 : b; \Delta, u_0 \leq u_1 \leq u_2 \vdash \varphi_0 \silent \)

\[ u_0 : b, u_2 : b; i : b; \Theta; \Sigma; \Gamma^L, u_0 : b, u_3 : b; i : b; \varphi_0 \vdash c_1 : x : \tau, \varphi_1 \]

\[ u_0, u_3 : b; i : b; \Theta; \Sigma; \Gamma^L, u_1 : b, u_3 : b; \varphi_0 \vdash c_1 : x : \tau, \varphi_1 \]

\[ u_1 : b, u_3 : b; i : b; \Theta; \Sigma; \Gamma^L, u_0 : b, u_3 : b; \Delta, u_0 \leq u_1 \leq u_3, \varphi_0 \vdash \varphi_2 c_2 : y : \tau, \varphi_2 \]

\( \Theta; \Sigma; \Gamma^L, u_0 : b, u_3 : b; i : b; \Gamma, u_1 : u_2 b, y : \tau; \Delta \vdash (\varphi_0 \land \varphi_1 \land \varphi_2) \Rightarrow \varphi \true \)

\( \Theta; \Sigma; \Gamma^L; u_0 : b, u_3 : b; i : b; \Gamma, y : \tau \vdash \varphi \ok \)

\[ u_0 : b, u_3 : b; i : b; \Theta; \Sigma; \Gamma^L; \Delta \vdash \varphi (e_1 ; c_2) : y : \tau, \varphi \]

\( \Theta; \Sigma; \Gamma^L, u_0 : b, u_3 : b; i : b; \Theta; \Sigma; \Gamma^L, u_3 : b; \Delta, u_0 \leq u_1 \leq u_2 \vdash \varphi_0 \silent \)

\[ u_0 : b, u_2 : b; i : b; \Theta; \Sigma; \Gamma^L, u_0 : b, u_3 : b; i : b; \varphi_0 \vdash c_1 : x : \tau, \varphi_1 \]

\[ u_1 : b, u_3 : b; i : b; \Theta; \Sigma; \Gamma^L, u_0 : b, u_3 : b; \varphi_0 \vdash c_1 : x : \tau, \varphi_1 \]

\[ u_1 : b, u_3 : b; i : b; \Theta; \Sigma; \Gamma^L, u_0 : b, u_3 : b; \Delta, u_0 \leq u_1 \leq u_3, \varphi_0 \vdash \varphi_2 c_2 : y : \tau, \varphi_2 \]

\( \Theta; \Sigma; \Gamma^L, u_0 : b, u_3 : b; i : b; \Gamma, u_1 : u_2 b, y : \tau; \Delta \vdash (\varphi_0 \land \varphi_1 \land \varphi_2) \Rightarrow \varphi \true \)

\( \Theta; \Sigma; \Gamma^L; u_0 : b, u_3 : b; i : b; \Gamma, y : \tau \vdash \varphi \ok \)

\[ u_0 : b, u_3 : b; i : b; \Theta; \Sigma; \Gamma^L; \Delta \vdash \varphi (e_1 ; c_2) : y : \tau, \varphi \]

\( \Theta; \Sigma; \Gamma^L, u_0 : b, u_3 : b; i : b; \Theta; \Sigma; \Gamma^L, u_3 : b; \Delta, u_0 \leq u_1 \leq u_2 \vdash \varphi_0 \silent \)

\[ u_0 : b, u_2 : b; i : b; \Theta; \Sigma; \Gamma^L, u_0 : b, u_3 : b; i : b; \varphi_0 \vdash c_1 : x : \tau, \varphi_1 \]

\[ u_1 : b, u_3 : b; i : b; \Theta; \Sigma; \Gamma^L, u_0 : b, u_3 : b; \varphi_0 \vdash c_1 : x : \tau, \varphi_1 \]

\[ u_1 : b, u_3 : b; i : b; \Theta; \Sigma; \Gamma^L, u_0 : b, u_3 : b; \Delta, u_0 \leq u_1 \leq u_3, \varphi_0 \vdash \varphi_2 c_2 : y : \tau, \varphi_2 \]

\( \Theta; \Sigma; \Gamma^L, u_0 : b, u_3 : b; i : b; \Gamma, u_1 : u_2 b, y : \tau; \Delta \vdash (\varphi_0 \land \varphi_1 \land \varphi_2) \Rightarrow \varphi \true \)

\( \Theta; \Sigma; \Gamma^L; u_0 : b, u_3 : b; i : b; \Gamma, y : \tau \vdash \varphi \ok \)

\[ u_0 : b, u_3 : b; i : b; \Theta; \Sigma; \Gamma^L; \Delta \vdash \varphi (e_1 ; c_2) : y : \tau, \varphi \]

**Figure 11.** Sequential composition

\[
\mathcal{EF}^D[u_b, u_e, i, \varphi]_{T, w} = \{(k, c) \mid \forall e. (k, e) \in \mathcal{EF}^D[u_b, u_e, i, \varphi]_{T, w} \implies \}
\]

\[ (k, c) \in \mathcal{EF}^D[u_b, u_e, i, \varphi]_{T, w} \]

**Semantics for invariant indexed types** Figure 12 summarizes the interpretation of types indexed by the invariant property \( u_b, u_e, i, \varphi \).

The invariant property is used to constrain the behavior of expressions that evaluate to normal forms that do not agree with their types.

\[
\mathcal{R}(u_b, u_e, i, \varphi)[x : \varphi \vdash \theta]_{T, w, i, \varphi} = \{(k, c) \mid \}
\]

\( j_x \) is the length of the trace from time \( u_2 \) to the end of \( T \)

\( k \geq j_x \geq j_x \)

the configuration at time \( u_1 \) is \( u_1 \vdash \sigma_b \cdots (i, x, c' :: K; c) \cdots \)

between \( u_1 \) and \( u_2 \) (inclusive), the stack of thread \( i \) always contains prefix \( x, c' :: K \)

\( \Rightarrow T \vdash \emptyset \)
Proof (sketch):

Indexed types are confined

1. \( k, a \in RC \{ |\forall k, c. u \langle x, \ast \rangle \in RA \} \Rightarrow (j, e) \in \mathcal{R}(u_b, u_e, i, \varphi)[\tau, \theta, T, u] \)

Formula semantics

\[
\begin{align*}
\forall X. \alpha & \Rightarrow \forall X. \alpha \\
\exists X. \alpha & \Rightarrow \exists X. \alpha \\
\forall x. \alpha & \Rightarrow \forall x. \alpha \\
\exists x. \alpha & \Rightarrow \exists x. \alpha \\
\end{align*}
\]

F. Lemmas

Lemma 5 (\( \mathcal{R}_{\mathcal{N}V} \) is downward-closed).

1. If \( (k, c) \in \mathcal{R}_{\mathcal{N}V}[\phi][\tau, u] \), then \( \forall j < k, (j, c) \in \mathcal{R}_{\mathcal{N}V}[\phi][\tau, u] \).
2. If \( (k, c) \in \mathcal{R}_{\mathcal{N}V}[\phi][\theta, T, u] \), then \( \forall j < k, (j, c) \in \mathcal{R}_{\mathcal{N}V}[\phi][\theta, T, u] \).
3. If \( (k, c) \in \mathcal{R}_{\mathcal{N}V}[\phi][\tau, u] \), then \( \forall j < k, (j, c) \in \mathcal{R}_{\mathcal{N}V}[\phi][\tau, u] \).

Proof (sketch): By examining the definition of the relations.

Lemma 6 (\( \mathcal{R}_{\mathcal{N}V} \) is closed under delay).

1. If \( (k, c) \in \mathcal{R}_{\mathcal{N}V}[\phi][\tau, u] \), then \( \forall u' > u, (k, c) \in \mathcal{R}_{\mathcal{N}V}[\phi][\tau, u] \).
2. If \( (k, c) \in \mathcal{R}_{\mathcal{N}V}[\phi][\theta, T, u] \), then \( \forall u' > u, (k, c) \in \mathcal{R}_{\mathcal{N}V}[\phi][\theta, T, u] \).
3. If \( (k, c) \in \mathcal{R}_{\mathcal{N}V}[\phi][\tau, u] \), then \( \forall u' > u, (k, c) \in \mathcal{R}_{\mathcal{N}V}[\phi][\tau, u] \).

Proof (sketch): By examining the definitions.

Lemma 7 (Indexed types are confined). \( \text{confine} (\tau) (u_b, u_e, i, \varphi) \)

1. \( \mathcal{R}(u_b, u_e, i, \varphi)[\tau][\theta, T, u] = \mathcal{R}(u_b, u_e, i, \varphi)[\theta, T, u] \).
2. \( \mathcal{R}(u_b, u_e, i, \varphi)[\tau][\theta, T, u] = \mathcal{R}(u_b, u_e, i, \varphi)[\theta, T, u] \).
3. for all \( n, c, \forall u_b, u_e, i. s.t. u \leq u_b \leq u_e, (n, c) \in \mathcal{R}(u_b, u_e, i, \varphi)[\tau, \varphi, u_b, u_e, i][\theta, T, u, G, I] \).

Proof. By induction on \( \tau \). 2 uses 1 directly, 1 uses 2 when \( \tau \) is smaller, 3 uses 2 directly, and 1 uses 3 when \( \tau \) is smaller. Proof of 1.

Lemma 8 (Invariance confinement).

\( \varphi \) is composable, and thread \( l \) is silent between time \( u_b \) and \( u_e \) implies \( T \vdash \varphi[l, u_b, u, i, \cdots, u_e, i] \).

1. If \( \text{fa}(c) = 0 \), \( \text{fc}(c) \in \text{Dom}(\gamma) \), \( (n, c) \in \mathcal{R}[u_b, u_e, i][\tau, \theta, T, u] \)

2. If \( \text{fa}(c) = 0 \), \( \text{fc}(c) \in \text{Dom}(\gamma) \), \( (n, c) \in \mathcal{R}[u_b, u_e, i][\tau, \theta, T, u] \)
G. Properties of Interpretation of Types

Lemma 9. If nf ≠ λ.x.e or λA.x.c or comp(c), then (n, nf) ∈ RV (Φ)[τ]Θ,T,u
Proof (sketch): Case on τ. For all cases except when τ = X, the conclusion follows from the definition of RV (Φ)[τ]Θ,T,u.
When τ = X, Θ(X) ∈ Type. By the definition of Type, every C ∈ Type contains all stuck terms that are not functions or suspended computations.

Lemma 10 (Substitution) If C = RV (Φ)[τ1]Θ1,T,u then
1. RV (Φ)[τ0(X → C)]Θ,T,u = RV (Φ)[τ1/X]Θ,T,u
2. RV (Φ)[τ0(X → C)]Θ,T,u = RV (Φ)[τ1/X]Θ,T,u
3. RV (Φ)[τ0[X → C]]Θ,T,u = RV (Φ)[τ1/X]Θ,T,u
4. RV (Φ)[n(X → C)]Θ,T,u = RV (Φ)[n(X)τ1/Θ1,T,u
Proof (sketch): By induction on the structure of τ, η, ϕ and α.

Lemma 11 (Downward-closure). If C ∈ RV (Φ)[τ]Θ,T,u then ∀ V ∈ (Θ, V) ∈ Type, (n, γ) ∈ RG (Φ)[Γ]Θ,T,u and j < n then (j, γ) ∈ RG (Φ)[Γ]Θ,T,u
Proof (sketch): By induction on the structure of Γ, using Lemma 11.

Lemma 12 (Substitutions are closed under index reduction). If ftr (Γ) ⊆ dom (Θ), ∀ V ∈ (Θ, V) ∈ Type, (n, γ) ∈ RG (Φ)[Γ]Θ,T,u and j < n then (j, γ) ∈ RG (Φ)[Γ]Θ,T,u
Proof (sketch): By Lemma 11.

Lemma 13 (Validity of types). If ftr (τ) ⊆ dom (Θ) and ∀ V ∈ (Θ, V) ∈ Type, then RV (Φ)[τ]Θ,T,u ∈ Type
Proof (sketch): By Lemma 11.

Lemma 14 (Closed under delay). If (k, e) ∈ RV (Φ)[τ]Θ,T,u and u' > u then (k, e) ∈ RV (Φ)[τ]Θ,T,u'
Proof (sketch): By Lemma 11.

Lemma 15 (Substitutions are closed under delay). If (n, γ) ∈ RG (Φ)[Γ]Θ,T,u and u' > u then (n, γ) ∈ RG (Φ)[Γ]Θ,T,u'
Proof (sketch): By induction on the structure of Γ, using Lemma 11.

H. Soundness

Theorem 16 (Soundness). Assume that ∀ A :: α ∈ Σ, ∀ Φ, T, n, u, (n, A) ∈ RA (Φ)[α]Θ,T,u then
1. (a) E :: u : b; θ; Σ; Γ; Δ ⊢ ϕ : e; τ
   • ∀ θ ∈ RT[Θ],
   • ∀ γ, η ∈ Π[Γ]
   • ∀ V, U, U', U' ≥ U, let γu = [U/u]
   • ∀ V, Σ, n, γ; (n, γ) ∈ RG (Φ)[Γ]Θ,T,u
   • T ⊢ Δγ, γ, τ
   • implies (n, cγ) ∈ RA (Φ)[τ]Θ,T,u
(b) E :: u, u2, i; θ; Σ; Γ; Δ ⊢ e : η
   • ∀ u, u2, u2, i.s.t. u ≤ u2 ≤ uE, let γu = [u2, u2, u/u2, i, u2, i]
   • ∀ θ ∈ RT[Θ],
   • ∀ γ ∈ Π[Γ]
   • ∀ V, Σ, n, γ; (n, γ) ∈ RG (Φ)[Γ]Θ,T,u
   • T ⊢ Δγ, γ, τ
   • implies (n, cγ) ∈ RA (Φ)[τ]Θ,T,u
Proof. By induction on the structure of \( \mathcal{E} \).

Proof of 1(a).

\[ \mathcal{E}': \forall \alpha \gamma \gamma \gamma \eta \in \mathcal{F}(\Phi) \forall \gamma \gamma \gamma \eta \in \mathcal{T}, \mathcal{E} \vdash \eta \mathcal{E} \]

case: Confine

\[ \varphi \text{ is trace composable} \]

\[ \mathcal{E}': \eta \mathcal{E} \vdash \eta \mathcal{E} \]

By assumptions

\[ \mathcal{E} \vdash \eta \mathcal{E} \]

By I.H. on \( \mathcal{E} \).

\[ \mathcal{E} \vdash \eta \mathcal{E} \]

By Lemma 12 and 15

\[ \mathcal{E} \vdash \eta \mathcal{E} \]

By I.H. on \( \mathcal{E} \).

By Lemma 12 and 15

\[ \mathcal{E} \vdash \eta \mathcal{E} \]

By (2) and (3)

\[ \mathcal{E} \vdash \eta \mathcal{E} \]

By I.H. on \( \mathcal{E} \).

\[ \mathcal{E} \vdash \eta \mathcal{E} \]

By Lemma 12 and 15

\[ \mathcal{E} \vdash \eta \mathcal{E} \]

By (2) and (3)

\[ \mathcal{E} \vdash \eta \mathcal{E} \]

By I.H. on \( \mathcal{E} \).

By Lemma 12 and 15

\[ \mathcal{E} \vdash \eta \mathcal{E} \]

By (2) and (3)

\[ \mathcal{E} \vdash \eta \mathcal{E} \]

By I.H. on \( \mathcal{E} \).

By Lemma 12 and 15

\[ \mathcal{E} \vdash \eta \mathcal{E} \]

By (2) and (3)

\[ \mathcal{E} \vdash \eta \mathcal{E} \]

By I.H. on \( \mathcal{E} \).

By Lemma 12 and 15

\[ \mathcal{E} \vdash \eta \mathcal{E} \]

By (2) and (3)

\[ \mathcal{E} \vdash \eta \mathcal{E} \]

By I.H. on \( \mathcal{E} \).

By Lemma 12 and 15

\[ \mathcal{E} \vdash \eta \mathcal{E} \]

By (2) and (3)

\[ \mathcal{E} \vdash \eta \mathcal{E} \]

By I.H. on \( \mathcal{E} \).

By Lemma 12 and 15

\[ \mathcal{E} \vdash \eta \mathcal{E} \]

By (2) and (3)

\[ \mathcal{E} \vdash \eta \mathcal{E} \]

By I.H. on \( \mathcal{E} \).

By Lemma 12 and 15

\[ \mathcal{E} \vdash \eta \mathcal{E} \]

By (2) and (3)

\[ \mathcal{E} \vdash \eta \mathcal{E} \]

By I.H. on \( \mathcal{E} \).

By Lemma 12 and 15

\[ \mathcal{E} \vdash \eta \mathcal{E} \]

By (2) and (3)

\[ \mathcal{E} \vdash \eta \mathcal{E} \]

By I.H. on \( \mathcal{E} \).

By Lemma 12 and 15

\[ \mathcal{E} \vdash \eta \mathcal{E} \]

By (2) and (3)

\[ \mathcal{E} \vdash \eta \mathcal{E} \]

By I.H. on \( \mathcal{E} \).

By Lemma 12 and 15

\[ \mathcal{E} \vdash \eta \mathcal{E} \]

By (2) and (3)

\[ \mathcal{E} \vdash \eta \mathcal{E} \]

By I.H. on \( \mathcal{E} \).

By Lemma 12 and 15

\[ \mathcal{E} \vdash \eta \mathcal{E} \]

By (2) and (3)

\[ \mathcal{E} \vdash \eta \mathcal{E} \]

By I.H. on \( \mathcal{E} \).

By Lemma 12 and 15

\[ \mathcal{E} \vdash \eta \mathcal{E} \]

By (2) and (3)

\[ \mathcal{E} \vdash \eta \mathcal{E} \]

By I.H. on \( \mathcal{E} \).

By Lemma 12 and 15

\[ \mathcal{E} \vdash \eta \mathcal{E} \]

By (2) and (3)

\[ \mathcal{E} \vdash \eta \mathcal{E} \]

By I.H. on \( \mathcal{E} \).

By Lemma 12 and 15

\[ \mathcal{E} \vdash \eta \mathcal{E} \]

By (2) and (3)

\[ \mathcal{E} \vdash \eta \mathcal{E} \]

By I.H. on \( \mathcal{E} \).

By Lemma 12 and 15

\[ \mathcal{E} \vdash \eta \mathcal{E} \]

By (2) and (3)

\[ \mathcal{E} \vdash \eta \mathcal{E} \]

By I.H. on \( \mathcal{E} \).

By Lemma 12 and 15

\[ \mathcal{E} \vdash \eta \mathcal{E} \]

By (2) and (3)

\[ \mathcal{E} \vdash \eta \mathcal{E} \]

By I.H. on \( \mathcal{E} \).

By Lemma 12 and 15

\[ \mathcal{E} \vdash \eta \mathcal{E} \]

By (2) and (3)

\[ \mathcal{E} \vdash \eta \mathcal{E} \]

By I.H. on \( \mathcal{E} \).

By Lemma 12 and 15

\[ \mathcal{E} \vdash \eta \mathcal{E} \]

By (2) and (3)

\[ \mathcal{E} \vdash \eta \mathcal{E} \]

By I.H. on \( \mathcal{E} \).

By Lemma 12 and 15

\[ \mathcal{E} \vdash \eta \mathcal{E} \]

By (2) and (3)

\[ \mathcal{E} \vdash \eta \mathcal{E} \]

By I.H. on \( \mathcal{E} \).

By Lemma 12 and 15

\[ \mathcal{E} \vdash \eta \mathcal{E} \]

By (2) and (3)

\[ \mathcal{E} \vdash \eta \mathcal{E} \]
case: $\text{SEQ}$

$$
\begin{align*}
\mathcal{E}_1 &::= u_0, u_1, i; \Theta; \Sigma; \Gamma^L; ; u_3; \Delta, u_0 \leq u_1 \vdash \varphi_0 \text{ silent} \\
\mathcal{E}_2 &::= u_1, u_2, i; \Theta; \Sigma; \Gamma^L; ; u_0 : b; u_3; \Delta, u_1 \leq u_2, \varphi_0 \\
\mathcal{E}_3 &::= \mathcal{E}_2, u_3, i; \Theta; \Sigma; \Gamma^L; , \mathcal{E}_1, \Delta, x: \tau; u_2 \leq u_3, \varphi_0, \varphi_1 \\
\mathcal{E}_4 &::= \mathcal{E}_3, \mathcal{E}_1, u_1, u_2, u_0, i; \Theta; \Sigma; \Gamma^L, x: \tau, y: \tau', \Delta \\
&\vdash (\varphi_0 \wedge \varphi_1 \wedge \varphi_2) \Rightarrow \varphi \text{ true} \\
&\vdash \mathcal{E}_0 \subseteq \mathcal{E}_1 \\
&(n, \text{let}(c_1, c_2)) \in \text{dom}(\mathcal{E}_1) \\
&u_0, u_3, i; \Theta; \Sigma; \Gamma^L; ; \Delta \vdash \text{let}(c_1, c_2) : x: \tau'; \varphi
\end{align*}
$$

By assumption

- Pick time points $u, u_B, u_E$ and thread $i$, s.t. $u \leq u_B \leq u_E$, let $\gamma_1 = [u_B, u_E, i/u_0, u_3, i]$.
- Let $\mathcal{E}_0 = \{b; u_B, u_E, i/u_0, u_3, i\}$.
- The length of the trace from time $u_B$ to the end of $T$ is $j_0$.
- The length of the trace from time $u_B$ to the end of $T$ is $j_e$.
- The configuration at time $u_B$ is $u_B \vdash \sigma_0 \cdots ; (i; y.c :: K; \text{let}(e_1, c_1, c_2)) \cdots$
- The configuration at time $u_E$ is $u_E \vdash \sigma_e \cdots ; (i; K; c[e/y]) \cdots$
- and between $u_B$ and $u_E$ (inclusive), the stack of thread $i$ always contains prefix $y.c :: K$.

By the operational semantics

- exists $u_{m1}, u_{m2}, s.t. u = u_{m1} \leq u_{m2} \leq u_E$.
- the configuration at time $u_{m1}$ is $u_{m1} \vdash \sigma_{m1} \cdots ; (i; x.c :: K; c[1_i]) \cdots$
- the configuration at time $u_{m2}$ is $u_{m2} \vdash \sigma_{m2} \cdots ; (i; y.c :: K; c[2_i]) \cdots$

By (4)

- between time $u_B$ and $u_{m1}$, thread $i$ is silent.

By (1),

\[ T = (\Delta_{\gamma^L} \gamma_1, (u_0 \leq u_1) \gamma_1[u_{m1}, u_1]) \]

By (1)

\[ (j_0, \gamma_1) \in \mathcal{R}(\Phi)[\Gamma^L, \gamma_0[u_B, u_E, i/u_0, u_3, i]]_{\mathcal{E}_0, \mathcal{T}, u} \]

By I.H. on $\mathcal{E}_1$ and (5), (6) and (7)

\[ T = \varphi \gamma_{\gamma^L \gamma_1}[u_{m1}, u_1] \]

Let $\gamma_2 = \gamma_{\gamma^L \gamma_1}[u_{m1}, u_1]$

By (1) and Lemma [5] $u \leq u_{m1}$

\[ (j_0, \gamma_2) \in \mathcal{R}(\Phi)[\Gamma^L, \gamma_0[u_B, u_{m1}, u_{m2}, i/u_0, u_1, u_2, u_3, i]]_{\mathcal{E}_0, \mathcal{T}, u_{m1}} \]

By I.H. on $\mathcal{E}_1$ and (6), (8), (9)

\[ (n, \text{let}(c_1)) \in \mathcal{R}(\Phi)[(x: \tau, \varphi_1) \gamma_2 \gamma_3]_{\mathcal{E}_1, \mathcal{T}, u_{m1}, u_{m2}} \]

By (10),

let $j_{m2}$ be the length of the trace from time $u_{m2}$ to the end of $T$

\[ (j_{m2}, \gamma_3) \in \mathcal{R}(\Phi)[\Gamma^L \gamma_2 \gamma_3]_{\mathcal{E}_1, \mathcal{T}, u_{m1}, u_{m2}} \]

By Lemma [12] and $j_{m2} < n$
I. Proof Sketch of State Integrity for Memoir

We prove the correctness of a TPM based state continuity mechanism that closely follows Memoir [28].

Terms, Actions and Predicates

We first describe here the terms, actions and predicates that model the TPM functionality, cryptography and communication.

TPM functionality. The TPM is modeled by the following actions. The actions reset_pcr(p) and extend_pcr(p, h), respectively resets the state of the PCR p to some default value and extends the value of p with the value h. The action verify_pcr(p, h) checks if the state of PCR p is h, otherwise aborts. The action setNVRAMlocPerms(Nloc, p) tyes the permissions for NVRAM location Nloc to the current contents of the PCR p. The actions NVRAMwrite(Nloc, m) and NVRAMread(Nloc) respectively write the message m and read from the NVRAM. The action ll_enter() starts a new late launch session with computation e called on some arguments. A late launch session is modeled by a new thread that runs e with no other thread running in parallel. The action ll_exit() exits from a late launch session.

Cryptography. Symmetric encryption is modeled by the actions encrypt(k, m) and decrypt(k, c). Message authentication codes are modeled by mac(k, m) and verify_mac(k, m, m'). Hash functions are modeled by the action hash(m). A message m encrypted by a key k is denoted by the term ENCk(m). Similarly, a MAC of a message m with key k is denoted by MACk(m). A hash is represented by the term hash(m). The special term code_hash(c) refers to the textual reification of the computation e. The term hash_chain(m1, m2, · · · , mℓ) is syntactic sugar for the iterated hash hash(hash(hash(m1)||m2|| · · · ||mℓ)). Here, the term m1||m2|| · · · ||mℓ represents the concatenation of messages.

Communication. Communication is modeled by the send(m) receive() action. By default, messages are not authenticated, so we drop the send and receive respectively do not have a recipient and sender argument.

Flags. To state the overall state continuity property, we require three flags (service_init, service_try and service_invoke) which simply record the value of variables at a particular point.

Figure 14 contains our model for the Memoir system. The suspended computation runmodule is expected to run in a late launch session that models both the initialization and execution phase of Memoir. Lines 14-26 model the initialization phase and lines 28-40 model the execution phase. We only describe the initialization phase here and the execution phase proceeds similarly. During initialize the code for service is hashed into PCR 17. Subsequently, it is checked whether PCR 17 contains a hash chain starting with -1 and followed by a hash of the textual reification of runmodule. This ensures that a late launch session with runmodule was initiated. A symmetric key is then generated that acts as the encryption and MAC key for subsequent sessions of Memoir. Then, the permissions on Nloc, the NVRAM location allocated for the session is tied to the current value of PCR17. An initial history summary and the symmetric key are then written to the NVRAM location, and then the value of PCR 17 is extended with a dummy value so that Nloc cannot be read unless a new runmodule session is started. The service is then initiated to generate a state of the service that is then encrypted and MACed along with the history summary and sent to the adversary for persistent storage.

Predicates. Each action has a corresponding action predicate. All action predicates are listed in Figure 14. Every action predicate has an additional argument that corresponds to the thread that performed that action. The one exception is the action predicate ll.Enter, for which the first argument j is the thread corresponding to the late launch session.

Apart from action predicates, we have predicates which capture state. The predicate val_pcr(p, h) states that at time u, the value of the PCR p is the hash h. The predicate NVPerms(Nloc, p, h) states that the permissions on the NVRAM location Nloc are set to the value of the PCR p being the hash h. The predicate val_NVRAM(Nloc, m) @ u states that the NVRAM location Nloc contains the value m at time u.

We have some predicates about the structure of terms. The predicate hash_prefix(h1, h2) states that the hash chain h2 can be obtained by extending h1 with additional hashes.

Table: Action Predicates

| Action | Predicate |
|--------|-----------|
| reset_pcr(p) | ResetPCR(i, p) |
| extend_pcr(p, h) | ExtendPCR(i, p, h) |
| verify_pcr(p, h) | VerifyPCR(i, p, h) |
| setNVRAMlocPerms(Nloc, p) | SetNVPerms(i, Nloc, p) |
| NVRAMwrite(Nloc, m) | NVRAMWrite(i, Nloc, m) |
| NVRAMread(Nloc) | NVRead(i, Nloc, m) |
| ll_enter() | LLEnter(i) |
| ll_exit() | LLError(i) |
| encrypt(k, m) | Encrypt(i, k, m) |
| decrypt(k, m) | Decrypt(i, k, m) |
| mac(k, m) | MAC(i, k, m) |
| verify_mac(k, m, m') | verifyMAC(i, k, m, m') |
| hash(k, m) | Hash(i, k, m) |
| service_init(skey, service, state, Nloc) | service_init(i, skey, service, state, Nloc) |
| service_try(skey, service, state, Nloc) | service_try(i, skey, service, state, Nloc) |
| service_invoke(skey, service, state, state', Nloc) | service_invoke(i, skey, service, state, state', Nloc) |

Figure 15 summarizes the abbreviations we use.

Abbreviations and Definitions

The proof proceeds in four stages. Each step employs the rely-guarantee technique in the style of [14] to prove a particular invariant about executions of the system. At a high level, the four stages of the proof are as follows:

1. PCR Protection: We show that the value of pcr17 contains a certain measurement h only during late launch sessions running a session of Memoir.
runmodule =
  let snapshot =
    λ(state, summary, skey).
    enc_state ← act(encrypt(skey, service_state));
    auth ← act(mac (skey, (enc_state, freshness_tag)));
    ret(enc_state, freshness_tag, auth)
  let check_snapshot =
    λ((enc_state, freshness_tag, auth), request, history, skey).
    act(verify_mac (skey, (enc_state, freshness_tag), auth));
    freshness_tag' ← act(hash (freshness_tag)[request]);
    if(freshness_tag = history ∨ freshness_tag' = history).
    act(dec (skey, enc_state)).
    act(abort())
  let initialize =
    λ(service, Nloc).
    act(extend_pcr(pcr17, code_hash(service)));
    act(verify_pcr(pcr17, hash_sha1n(−1, code_hash(runmodule), code_hash(service))));
    skey ← act(gen_symkey());
    let history_summary = 0
    act(setNVRAMLocPerms(Nloc, pcr17));
    act(NVRAMWrite(Nloc, (history_summary, skey)));
    act((extend_pcr(pcr17, 0)));
    service_state ← (service ExtendPCR ResetPCR · · ·) INIT;
    act(service_init(skey, service, service_state, Nloc));
    snap ← snapshot(service_state, history_summary, skey);
    ret((), snap)
  let execute =
    λ(service, Nloc, snap, req).
    act(extend_pcr(pcr17, code_hash(service)));
    (skey, history_summary) ← act(NVRAMThread Nloc);
    service_state ← check_snapshot(snap, request, history_summary, skey);
    new_summary ← act(hash (history_summary)[req]);
    act(NVRAMWrite(Nloc, (new_summary, skey)));
    act(extend_pcr(pcr17, 0));
    act(service_try(skey, service, service_state, Nloc));
    (new_state, resp) ← (service ExtendPCR ResetPCR · · ·) (EXEC(service_state, req));
    snap ← snapshot(service_state, history_summary, skey);
    act(service_invoke(skey, service, service_state, new_state, Nloc));
    ret(resp, snap)
  λ(service, Nloc, call).
  (resp, snap) ← (case call of
    INIT ⇒ initialize(service, Nloc)
    EXEC(snap, req) ⇒ execute(service, Nloc, snap, req))
  act(send(response, snap));
  call(ll_exit())

Figure 13. runmodule: A model of Memoir’s state isolation mechanism

2. NVRAM Protection: We show that after the permissions on a location in the NVRAM has been set to h, then the permissions on that location are never changed.

3. Key Secrecy: We show that if the key corresponding to the service is available to a thread, then it must have either generated it or read it from the NVRAM.

4. History Summary-State Correspondence: We show that if on any two executions of the Memoir, if the history summaries are equal then the states must also be equal.

Finally, from these, we prove the overall state continuity property for Memoir.

Next, we sketch the proofs of each of the above stage. The proofs require axioms about the above predicates, which we state along with the stage the axioms are first required.

1.1.1 PCR Protection.

In Figure 15, we list the definitions and model specific axioms we need. The predicate InLLSession(u1, u2, e, j) states that thread j runs a late launch session for e between u1 and u2. The predicate InSomeLLSession(u, e) states that at time u, thread j runs a late launch session for e. The predicate InSomeLLSession(u, e) states that at time u, some thread is running a late launch session for e. LLThread(j, e) states that j is a thread that runs a late launch session for e. PCRPrefix(p, s_hash) states that the value contained in p is a hash prefix of s_hash. ExitsPCRProtected(i, u, s_hash) states
In 

ExitsPCRProtected

InLLSession

whenever a late launch thread exists, the state of PCR 17 is a prefix of code

ψ

ExitsPCRProtected

LLThread

LLEnter

val

(PCRInit)

∀

i,u,s

ψ

val

(1)

LLEnter(i,e) @ u ⇒ ExitsPCRProtected(i,e) @ u

PCRPREFIX(p,s_hash) = ∃h, val PCR (pcr17,h) ∧ hash_prefix(h,s_hash)

ExitsPCRProtected(i,u,s_hash) = LLEXIT(i) @ u ⇒ ¬PCRPREFIX(pcr17,s_hash) @ u

LLChain(h,e) = hash_prefix(hash_chain(-1,code_hash(e)),h)

Definitions

(1)

ExitsPCRProtected

Figure 16. Definitions and Model-specific axioms about late launch

that whenever a late launch thread exists, the state of PCR 17 is not a prefix of s_hash. LLChain(h,e) states that h is a hash chain, which if contained in PCR 17, is evidence of a late launch session for e.

Axiom (LLExit) states that whenever outside a late launch session, the value of PCR 17 is found to be a late launch chain s_hash, we can conclude, that some late launch session existed with the state of PCR 17 being a prefix of s_hash. (PCRInit) states that the value of any PCR begins at 0. (LLEnter) states that late launch threads for a computation e exclusively run e with some arguments e'. (LLAct1) and (LLAct2) are axiom schemas that essentially state that no other threads are active during late launch sessions.

Consider an arbitrary service s. Let s_hash = hash_chain(-1,code_hash(hash_module),code_hash(s)). We show that if the value of pcr17 at time u is s_hash, then it must be the case that we are in a late launch session at time u. Formally, we show that,

∀u.val PCR (pcr17,s_hash) @ u ⇒ InSOME LL Session(u, runmodule)

(1)

To prove an invariant ∀u > u', φ(u), using rely guarantee reasoning, it is sufficient to show for a choice of ψ(i,u) and i(i) that

(1)

φ(u)

(2) ∀i, u. (i(i) ∧ ∀u' < u. φ(u')) ⇒ ψ(i,u)

(φ(u1) ∧ ¬φ(u2) ∧ (u1 < u2)) ⇒ 3i, u3. (u1 < u3 ≤ u2) ∧ i(i) ∧ ¬ψ(u3,i) ∧ ∀u4 ∈ (u1,u3), φ(u4)

We choose ψ, ψ and i as below:

ψ(u) = val PCR (pcr17,s_hash) @ u ⇒ InSOME LL Session(u,runmodule)

ψ(i,u) = ExitsPCRProtected(i,u, s_hash)

i(i) = LLThread(i,runmodule)

Axioms

(PCRInit)

val PCR (p,0) @ −∞ ⇒ InSOME LL Session(u,e)

(LLEnter)

∀s_hash, u2,e

LLChain(s_hash,e) ⇒ val PCR (pcr17,s_hash) @ u2

∧ ¬InSOME LL Session(u2,e) ⇒ ∃j,u3.

LLThread(j,e) ∧ LLEXIT(j) @ u3

∧ val PCR (pcr17,h) @ u3

∧ hash_prefix(h,s_hash)

∧ ∀u ∈ (u1,u3).

val PCR (pcr17,s_hash) @ u ⇒ InSOME LL Session(u,e)

(1)

Figure 17. Model-specific axioms about NVRAM

Condition (1) follows (PCRInit) and ¬hash_prefix(0,s_hash). Condition (3) follows directly from axiom (LLExit). To prove conditions (2) above, expanding out the definitions of φ, i and ψ above, we need to show that

∀i,u. (LLThread(i,runmodule)

∧ ∀u' < u. (val PCR (pcr17,s_hash) @ u' ⇒ (InSOME LL Session(u,runmodule)(u'))) ⇒ ExitsPCRProtected(u,i)

This can be rewritten as

∀i. (LLThread(i,runmodule)

∧ ∀u. (∀u' < u. (val PCR (pcr17,s_hash) @ u' ⇒ (InSOME LL Session(u,runmodule))) ⇒ ExitsPCRProtected(u,i)

Choose an arbitrary thread i such that LLThread(i,runmodule). Therefore, we have by (LL Honest) that for some e', start(−∞,runmodule, e',i). To use rule HONEST to show (3), we need to show that runmodule satisfies the following invariant.

∀u < u'. (val PCR (pcr17,s_hash) @ u' ⇒ (InSOME LL Session(u,runmodule)) ⇒ ExitsPCRProtected(u,i)

The key step in typing runmodule is to type the execution of s supplied by the adversary using the CONFINE rule. Essentially, we need to show that the service cannot exit with the pcr17 containing a prefix of s_hash. The service is confined to the actions provided by the TPM and we can show that each of them has the following computational type expr(u0,uc,i,x.φc,φr), where φc is:

φc = ¬PCRPREFIX(pcr17,s_hash) @ u0 ⇒ ∀u ∈ [u0,uc].(InSOME LL Session(u,runmodule,i) ⇒ ¬PCRPREFIX(pcr17,s_hash) @ u)

(5)

1.1.2 NVRAM Protection

Figure 17 contains axioms governing the behavior of NVRAM.

SetPerms states that on the successful execution of setting permissions on NVRAM at time u, the permissions are correct at u. (GetPerms) states that when the permissions on a particular NVRAM location is tied to the PCR p being h, then accessing that NVRAM location implies that the value of PCR p is h. (NVRAMPPerms)
states that if the permissions on a NVRAM location changes, then it must have been changed via a SetNVRAMLocPerms action.

We wish to show that the permissions on the NVRAM are always tied to the value of pcr17 being $s_{\text{hash}}$:

\[
\text{SetNVRAMLocPerms}(i, Nloc, pcr17) \land \text{val}_{pcr}(pcr17, s_{\text{hash}}) @ u_i \Rightarrow \forall (u > u_i). \text{NVRAM}(Nloc, pcr17, s_{\text{hash}}) @ u
\]

Assume that for some time point $u_i$,

\[
\text{SetNVRAMLocPerms}(i, Nloc, pcr17) \land \text{val}_{pcr}(pcr17, s_{\text{hash}}) @ u_i
\]

We now need to show that

\[
\forall (u > u_i) \Rightarrow \text{NVRAM}(Nloc, pcr17, s_{\text{hash}}) @ u
\]  

Again, we prove this invariant by rely guarantee reasoning, where we choose $\varphi$, $\psi$ and $i$ to be the following.

\[
\begin{align*}
\varphi(u) &= \text{NVRAM}(Nloc, pcr17, s_{\text{hash}}) @ u \\
\psi(u, i) &= (\text{SetNVRAMLocPerms}(i, Nloc, p) @ u) \\
i(i) &= \text{LLThread}(i, \text{runmodule})
\end{align*}
\]

Expanding condition (1), we need to show the following:

\[
\text{NVRAM}(Nloc, pcr17, s_{\text{hash}}) @ u_i
\]

This holds by Axiom (SetPerms) and [4].

Expanding condition (2), choose $i$ such that LLThread($i$, runmodule).

We need to show that $\forall u > u_i, (\forall u' \in [u_i, u], \varphi(u')) \Rightarrow \psi(i, u)$.

To use HONEST, we need to show that runmodule satisfies the following invariant.

\[
\forall u \in [u_i, u], \forall u' \in [u_i, u]. \text{NVRAM}(Nloc, pcr17, s_{\text{hash}}) @ u' \Rightarrow \text{SetNVRAMLocPerms}(i, Nloc, p) @ u' \\
\Rightarrow \forall p. \text{val}_{pcr}(pcr17, s_{\text{hash}}) @ u
\]

Again, the key step in typing runmodule is to type the execution of $s$ supplied by the adversary using the CONFINE rule. Essentially, we show that the service is not allowed to set the permissions of $Nloc$ at all. Each action $f$ provided by the TPM interface can be confined to the type $\text{cmp}(u_b, u_e, i, \{x, \varphi_c, \varphi_e\})$, where $\varphi_c$ is:

\[
f : \text{cmp}(u_b, u_e, i, \text{PCRPrefix}(pcr17, s_{\text{hash}}) @ u_b) \Rightarrow \forall u \in [u_b, u_e]. (\text{InLLSess}(u, \text{runmodule}, i) \Rightarrow \forall p. \text{val}_{pcr}(pcr17, s_{\text{hash}}) @ u)
\]

Condition (3) follows from (NVRAM), (GetPerms) and [1].

In particular, we can show from [4] and (GetPerms):

\[
\text{SetNVRAMLocPerms}(i, Nloc, pcr17) \land \text{val}_{pcr}(pcr17, s_{\text{hash}}) @ u_i \\
\Rightarrow \forall (u > u_i). \text{ReadNV}(I, Nloc) @ u \\
\Rightarrow \text{val}_{pcr}(pcr17, s_{\text{hash}}) @ u
\]

And by [1]

\[
\text{SetNVRAMLocPerms}(i, Nloc, pcr17) \land \text{val}_{pcr}(pcr17, s_{\text{hash}}) @ u_i \\
\Rightarrow \forall (u > u_i) \Rightarrow \text{ReadNV}(I, Nloc) @ u \\
\Rightarrow \text{InSomeLLSess}(u, \text{runmodule})
\]

Therefore, by (LLAct), we have that

\[
\text{SetNVRAMLocPerms}(i, Nloc, pcr17) \land \text{val}_{pcr}(pcr17, s_{\text{hash}}) @ u_i \\
\Rightarrow \forall (u > u_i) \Rightarrow \text{ReadNV}(I, Nloc) @ u \\
\Rightarrow \text{InSomeLLSess}(u, \text{runmodule})
\]

## Definitions

- NVContains($Nloc, s$) = $3m_{\text{Contains}}(m, s) \land \text{val}_{NV}(m, s)$
- Private($s, Nloc, u$) = $\forall u' < u. (\text{Send}(i, m) @ u \Rightarrow \neg \text{Contains}(m, s) \land \neg \text{NVLoc}(\text{NVContains}(Nloc, s, i) @ u') \Rightarrow (\text{NLoc} = Nloc))$
- KeepsPrivate($i, Nloc, Nloc$) = $\text{Send}(i, m) \Rightarrow \text{Contains}(m, s) \land \text{NVLoc}(\text{WriteNV}(Nloc, m) \land \text{Contains}(m, s) \Rightarrow \text{NLoc} = Nloc')$
- NewInLL($s, e$) = $\text{New}(i, s) @ u \Rightarrow \text{InLLSess}(u, e, i)$

## Axioms

- (Shared) LLChain($h, e$) \land NewInLL($s, e$) \Rightarrow \forall u \in [u_1, u_2] \in [u_i, \infty]. \text{Private}(s, Nloc, u_1) \land \text{Private}(s, Nloc, u_2) \Rightarrow \exists i, u_3, u_4 < u \not\subset u_2$.
- (POS) $\text{Private}(s, Nloc, u) \land \text{Has}(i, s) @ u \Rightarrow (\exists u', u' < u \land \text{New}(i, s) @ u' \lor (\exists u', u' < u \land \text{ReadNV}(i, Nloc, m) @ u' \land \text{Contains}(m, s))$.
- (PrivateInit) $\text{New}(s, Nloc, u) \Rightarrow \text{Private}(s, Nloc, u)$.
- (New3) $\text{New}(s, Nloc, u) \land \text{New}(s', i) @ u \Rightarrow (i = i') \land (u = u')$.

## Figure 18. Definitions and Model-specific axioms about Secrecy

This means that whenever a thread $i$ reads from the $Nloc$ at time $u_i$, it must be the case that $i$ is in a late launch session running $\text{runmodule}$ at time $u_i$.

### 1.1.3 Key Secrecy

Figure 18 lists the definitions and axioms pertaining to key secrecy. The definition NVContains($Nloc, s$) states that the NVRAM location $Nloc$ contains the secret $s$. Private($s, Nloc, u$) states that the secret $s$ hash not been sent out on the network and the only NVRAM location it has been stored in is $Nloc$. KeepsPrivate($i, s, Nloc$) states that whenever a thread $i$ sends a message, it does not contain the secret $s$. Additionally, it only stores $s$ in $Nloc$. NewInLL($s, e$) states that $s$ was generated in a late launch session of $e$.

The axiom (Shared) states that if a secret is private at time $u_1$ and not private at $u_2$, then it must be the case, that at some point in the middle some thread violated KeepsPrivate($i, s, Nloc$). (POS) states that if some thread posses a secret $s$ that is private to $Nloc$, then it must have been either generated in that thread or read from $Nloc$. (PrivateInit) states that a secret is private as soon as it is generated. (New3) is an axiom about non-collision of nonce values. (Init) is a logical assumption we make that states that service_init can only be called by honest threads running runmodule.

We now show that after initialization, if any thread $j$ has the key corresponding to the service, then that thread must have read it from $Nloc$ or that the thread $j$ is the initialization thread itself.
∀i, u, state, skey, Nloc
\text{service_init}(i, \text{skey}, \text{service}, \text{state}, \text{Nloc})@u_i \Rightarrow
\forall j, u > u_i. \text{Has}(j, \text{skey})@u \Rightarrow (j = i) \lor
\exists u', m. (u_i < u' < u) \wedge \text{ReadNV}(j, \text{Nloc}, m}@u' \wedge \text{Contains}(m, \text{skey})
\tag{13}
\]
\]
Fix I_i, u, \text{skey}, \text{service}, \text{Nloc}.
Assume \text{service_init}(I_i, \text{skey}, \text{service}, \text{state}, \text{Nloc})@u_i
We prove I_i by another rely-guarantee proof, very similar to the proof of Kerberos in [14]. We choose the following \( \varphi, \psi \) and \( i \).
\[
\varphi(u) = \text{Private}(\text{skey}, \text{Nloc}, u)
\]
\[
\psi(i, u) = \text{KeepsPrivate}(i, \text{skey}, \text{Nloc})@u
\]
\[
i(i) = \text{LLThread}(i, \text{runmodule})
\]
To show condition (1): \( \varphi(u) \) we can first show using (Init), (HON) and reasoning about ordering and atomicity of events that:
\[
\exists u_1, u_2, u_3, u_4.(u_1 < u_2 < u_3 < u_4 < u_i)
\]
VerifyPCR(per17, s_hash}@u_i
\[
\text{New}(\text{skey})@u_2 \wedge
\text{SetNVPerms}(I_i, \text{Nloc}, \text{per17})@u_3 \wedge
\text{NVWrite}(I_i, \text{Nloc}, (\text{skey}, h))@u_4 \wedge
\neg\text{SetNVPerms}(I_i, \text{Nloc}, p) @ (u_3, u_i) \wedge
\neg(\text{Extend}(I_i, \text{per17}, l) \lor \text{Reset}(I_i, \text{per17})) @ (u_2, u_3)
\]
\[
\neg\text{Send}(I_i, m) @ (u_1, u_2)
\tag{14}
\]
Now we can show using (Shared), (PrivateInit), (LLAct) and (HON) that \( \text{Private}(\text{skey}, \text{Nloc}, u_i) \) holds. Essentially, at \( u_i, s \) is still private because the thread \( I_i \) did not leak the key, and no other thread was running in parallel.

To prove condition (2) we again use the HON rule. However, the property required is not derived using CONFINE. The key step is to show that if the service, which is untrusted code, is not given the key as an input, then it cannot leak the key during execution. We do this by assuming that the original service that Memoir was initialized with had this property and then prove that the service passed into any session of Memoir has to be equal to the service was initialized with. This is where we require the Eq rule to be used.

The key step here is the typing of the execution of \( s \)
\[
(s \text{ ExtendPCR} \text{ ResetPCR} \cdots) \text{ EXEC} (\text{service state, req})
\]
Here, we use the Eq rule As we can show that \( s = \text{service} \), by comparing the hash chains in PCR 17, we assign \( s \) the following type:
\[
(s \text{ ExtendPCR} \text{ ResetPCR} \cdots) : \text{II}_i : \text{msg}. \text{cmp}(u_i, u_e, i).
\]
\[
(x : \text{msg}. \neg\text{Contains}(i, s) \Rightarrow \neg\text{Contains}(x, s), \text{KeepsSecret}(i, \text{skey}, \text{Nloc}) @ [u_i, u_e])
\tag{15}
\]
Condition (3) Follows from (Shared), and (12).

I.1.4 State to History Summary Correspondence.

We state without proof an invariant that the history summary has a one-to-one correspondence with the state. This is proved through an induction on the history summary.
\[
\forall i, u, \text{state}, \text{skey}, \text{Nloc}
\text{service_init}(i, \text{skey}, \text{state}, \text{Nloc}, \ldots)@u_i \Rightarrow
\forall h, \text{state}, j, j' > u_i \wedge \text{mac}(j, \text{skey}, (\text{state}, h))@u \wedge \text{mac}(j', \text{skey}, (\text{state'}, h))@u' \Rightarrow (\text{state} = \text{state'})
\tag{16}
\]

I.1.5 State Continuity

The property we prove about Memoir is as follows:
\[
\forall u_i, \text{state}, \text{state'}, \text{skey}, \text{i_init}, \text{s_init}
\text{service_init}(\text{i_init}, \text{skey}, \text{service}, \text{s_init})@u_i \Rightarrow
\forall u > u_i, \text{service}_\text{try}(i, \text{skey}, \text{state})@u_i
\Rightarrow
\exists u', \neg\text{service}_{\text{try}}(j, \text{skey}, \text{state}@u' \wedge
\text{service}_{\text{try}}(j, \text{skey}, \text{state}@u') \wedge
\neg\text{service}_{\text{try}}(j, \text{skey}, \text{state}@u'' \Rightarrow (u', u'') @ (u', u''))
\tag{17}
\]
In the above statement, we elide unnecessary arguments in the flag predicates. This property states that for every execution attempt of the service with state \( \text{state} \) at time \( u \), there exists a prior time point \( u' \) such that at \( u' \) either (1) service was invoked resulting in state \( \text{state} \), or (2) there was an execution attempt of the service with state \( \text{state'} \) or (3) the service was initialized with state \( \text{state} \). Additionally, since \( u' \), the service has not been invoked, which would have advanced the state of the service. This last clause rules out any rollback attacks. Each flag is indexed with the same secret key \( \text{skey} \) that the service was initialized with. This key ties all the flags in the property to to the same instance of Memoir.

Fix an \( i, u, \text{state}, \text{skey} \),
Assume \text{service_init}(\text{i_init}, \text{skey}, \text{service}, \text{s_init})@u_i
For some \( u > u_i \) assume that
\[
\text{service}_{\text{try}}(i, \text{skey}, \text{state}, \text{state'})@u_i.
\tag{18}
\]
Therefore we have \( \text{Has}(i, \text{skey})@u_i \). By (13) we have that one of the two hold
\[
i = i_\text{init} \lor
\exists u'. u_i < u' < u. \text{ReadNV}(i, \text{Nloc}, m}@u' \wedge \text{Contains}(m, \text{skey})
\tag{19}
\]
We analyze each case:

Case 1
We have from (Init) and \text{service_init}(\text{i_init}, \text{skey}, \text{service}, \text{s_init}) that
\[
\exists u.(u < u_i) \land \text{Start}(i, \text{runmodule service Nloc INIT}@u
\]
With HONEST, we can show that \text{service}_{\text{try}} does not occur on \( i \) and we have a contradiction.

Case 2
\[
\exists u' \in (u_i, u). \text{ReadNV}(i, \text{Nloc}, m}@u' \wedge \text{Contains}(m, \text{skey})
\]
In this case, by (12) We have that LLThread(j, runmodule)
Therefore, by HONEST
\[
\text{ReadNV}(i, \text{Nloc}, (\text{skey}, h))@u'
\tag{20}
\]
By (NVRAMRead), we have that \( \exists u'' < u \) such that
\[
\text{WriteNV}(j, \text{Nloc}, (\text{skey}, h))@u'' \wedge
\forall j'', \neg\text{ReadNV}(j'', \text{Nloc}, m') @ (u'', u')
\tag{21}
\]
Again, by (12) and (21) we have that
LLThread(j, runmodule)
(22)
And by HONEST, as we know (21), we can derive that
\[
\text{mac}(j, \text{skey}, (\text{ENC}_{\text{skey}}(\text{state'}, h))
\tag{23}
\]
Also, from (18) and HONEST we know that the branch at Line 12 of runmodule executed. This gives us two cases:

Case 1:
\[
\text{verifyMAC}(i, \text{skey}, (\text{ENC}_{\text{skey}}(\text{state}, h))
\tag{24}
\]
This is the case where the history summary \( h \) matches the MACed history summary. From (24) and (MAC), we have for some \( j' \)

\[
\text{mac}(j', skey, (ENC_{skey}(\text{state}), h))
\]  

(25)

By (16) along with (24) and (26), we have \( \text{state}' = \text{state} \)
We then have from (23) that there exists a \( u' \) such that

\[
\text{service\_invoke}(j, skey, s', \text{state})@u' \\
\lor \text{service\_init}(j, skey, \text{service}, \text{state})@u'
\]  

(26)

Also, from (21), we can show that \( \forall j'' \lor \text{service\_invoke}(j'', \ldots) \).

- **Case 2:**

\[
\text{verifyMAC}(j, skey, (ENC_{skey}(\text{state}), h') \land h = H(\text{req}||h'))
\]  

(27)

This is the case where at Line 12 of runmodule, the current history summary is the hash of the current request and the history summary in the snapshot. This means that Memoir was called with exactly the same request in the past and no other request has completed since then. This case proceeds similarly to Case 1.