REPORT DOCUMENTATION PAGE

Form Approved OMB NO. 0704-0188

1. REPORT DATE (DD-MM-YYYY)	2. REPORT TYPE	3. DATES COVERED (From - To)
	Technical Report	-

4. TITLE AND SUBTITLE
System M: A Program Logic for Code Sandboxing and Identification

5a. CONTRACT NUMBER
W911NF-09-1-0273

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
611102

6. AUTHORS
Limin Jia, , Shayak Sen, , Deepak Garg, , Anupam Datta

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAMES AND ADDRESSES
Carnegie Mellon University
5000 Forbes Avenue

Pittsburgh, PA 15213 -3815

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS (ES)
U.S. Army Research Office
P.O. Box 12211
Research Triangle Park, NC 27709-2211

10. SPONSOR/MONITOR'S ACRONYM(S)
ARO

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
56390-CS.333

12. DISTRIBUTION AVAILIBILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
The views, opinions and/or findings contained in this report are those of the author(s) and should not contrued as an official Department of the Army position, policy or decision, unless so designated by other documentation.

14. 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 sound- ness of System M

15. SUBJECT TERMS
program logic, traces, adversarial code, security

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	15. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Pradeep Khosla
UU	UU	UU	UU		19b. TELEPHONE NUMBER 412-268-5090

Report Title

System M: A Program Logic for Code Sandboxing and Identification

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 sound- ness 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.

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.

Keywords Program logic, traces, adversarial code, security

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, en-

able 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 in a dynamic check (e.g., in secure boot [27] 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.* [14] is that in this paper trusted interfaces can receive and execute 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 adversarial code be checked in a rich type system prior to execution, which eliminates the need for a rule like CONFINE. Section 3 explains intuitions behind these two principles in more detail.

We show soundness of System M relative to a step-indexed model [2] built over syntactic traces. As in some prior work [8–10, 14], our semantics of assertions and postconditions account for interleaving actions from concurrently executing programs including adversarial programs and, hence, our soundness theorem implies that all verified properties hold in the presence of adversaries, which is a variant of robust safety, proposed by Gordon et al. [15]. System M supports *compositional proofs*—security proofs of sequentially composed programs are built from proofs of their sub-programs. System M also admits concurrent composition—properties proved of a program hold when that program executes concurrently with other, even adversarial, programs.

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 β-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 [28], a previously proposed trusted computing system. For reasons of space, we elide proofs, some technical details and several typing rules from this paper. These are presented in the 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,

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

Figure 1. Term Syntax

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 ($\texttt{tt}, \texttt{ff}$), natural numbers ($n \in \mathcal{N}$), thread identifiers ($\iota \in \mathcal{I}$), and memory locations ($\ell \in \mathcal{L}$). We use $\cdot$ as the place holder for the type in a polymorphic function instantiation. Suspended computations $\texttt{comp}(c)$ constitute a monad with return $\texttt{ret}(e)$ and bind $\texttt{lete}(e_1, x.c_2)$.

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.

A basic computation is either an atomic action ($\texttt{act}(a)$) or $\texttt{ret}(e)$ that returns the pure expression e immediately. $\texttt{fix}\, f(x).c$ is a fixpoint operator. f, which represents a suspended fixpoint computation, may appear free in the body c. Computation $(c\ e)$ is the application of a fixpoint computation to its argument. $\texttt{letc}(c_1, x.c_2)$ denotes the sequential composition of c_1 and c_2, while $\texttt{lete}(e_1, x.c_2)$ is the sequential composition of the suspended computation to which e_1 reduces and c_2. In both cases, the expression returned by the first computation is bound to x, which may occur free in c_2. We sometimes use the alternate syntax $x \leftarrow c_1; c_2$ and $\texttt{let}\ x = e_1; c_2$. When the expression returned by the first computation is not used c_2, we write $c_1; c_2$ and $e_1; c_2$.

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

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

A thread T is a unit of sequential execution. A non-stuck thread is a triple $\langle \iota; K; c \rangle$ or $\langle \iota; K; e \rangle$, where ι is a unique identifier of that thread (drawn from a set $\mathcal{I}$ of such identifiers), K is the execution (continuation) stack, and c and e are the computation and expression currently being evaluated. A thread permanently enters a stuck state, denoted $\langle \iota; \mathsf{stuck} \rangle$, after performing an illegal action, such as accessing an unallocated memory location. An execution stack is a list of frames of the form $x.c$ recording the return points of sequencing statements in the enclosing context. In a frame $x.c$, x binds the return expression of the computation preceding c. A configuration of the system is a shared state σ and a set of all threads. σ 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 β-reduction $\rightarrow_\beta$. This choice simplifies the operational semantics and the soundness proofs, as explained in Sections 6. We elide the standard rules for $\rightarrow_\beta$. The small-step transitions for threads and system configurations are shown in Figure 2. The relation $\sigma \triangleright T \hookrightarrow \sigma' \triangleright T'$ defines a small-step transition of a single thread. $\mathcal{C} \rightarrow \mathcal{C}'$ denotes a small-step transition for configuration $\mathcal{C}$; it results from the reduction of any single thread in $\mathcal{C}$.

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

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

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

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

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

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

$$\frac{}{\sigma \triangleright \langle \iota; x.c_2 :: K; \mathtt{comp}(c_1)\rangle \hookrightarrow \sigma \triangleright \langle \iota; x.c_2 :: K; c_1\rangle}\ \text{R-SEQE3}$$

$$\frac{}{\begin{array}{l}\sigma \triangleright \langle \iota; K; (\mathtt{fix} f(x).c)\ e\rangle \\ \hookrightarrow \sigma \triangleright \langle \iota; K; c[\lambda z.\mathtt{comp}(\mathtt{fix}(f(x).c)\ z)/f][e/x]\rangle\end{array}}\ \text{R-FIX}$$

Figure 2. Selected small-step reduction semantics of configurations

The rules for $\sigma \triangleright T \hookrightarrow \sigma' \triangleright T'$ are mostly straightforward. The rules for evaluating an atomic action (R-ACTS and R-ACTF) rely on a function next that takes the current store σ and an action a, and returns a new store and an expression, which are the result of the action. If the action is illegal, then $\mathsf{next}(\sigma, a) = (\sigma', \mathsf{stuck})$. If the action returns a non-stuck expression e (rule R-ACTS), then the top frame $(x.c)$ is popped off the stack, and $c[e/x]$ becomes the current computation of the thread. If next returns stuck (rule R-ACTF), then the thread enters the stuck state and permanently remains there. When a sequencing statement $\mathtt{lete}(e_1, x.c_2)$ is evaluated, the frame $x.c_2$ is pushed onto the stack, and e_1 is first reduced to a suspended computation $\mathtt{comp}(c_1)$; then c_1 is evaluated. When a fixpoint $(\mathtt{fix} f(x).c)$; e is evaluated, f is substituted with a function whose body is a suspension of $\mathtt{fix} f(x).c$.

Any *finite* execution of a configuration results in a trace $\mathcal{T}$, defined as a finite sequence of reductions. With each reduction we associate a time point u, also called a (logical) *time point*. These time points on the trace are monotonically increasing. A trace annotated with time is written $\xrightarrow{u_0} C_0 \xrightarrow{u_1} C_1 \ldots \xrightarrow{u_n} C_n$, where $u_i \leq u_{i+1}$. We follow the convention that the reduction from C_i to C_{i+1} happens at time u_{i+1} and that its effects occur immediately. Thus the state at time u_i is the state in C_i.

3. Motivating Application

We briefly review Memoir [28], our main application, and highlight the challenges in analyzing Memoir to motivate the novel typing rules for deriving properties of adverary-supplied code using code identification and sandboxing.

3.1 Overview of Memoir

Memoir provides state-integrity guarantees for stateful security-sensitive services invoked by potentially malicious parties. Such services often rely on untrusted storage to store their persistent state. An example of such a service is a password manager that responds with a stored password when it receives a request containing a URL and a username. 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 the attacker from invoking the service with a valid but old state, and consequently mounting service rollback 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:

```
1  runmodule(srvc, snap, req, Nloc) =
2    ...
3    (skey, freshness_tag) ← act(NVRAMread Nloc);
4    service_state ← check_decrypt_snapshot (snap);
5    ...
6    (state′, resp)
         ← (srvc ExtendPCR ResetPCR ···) (state, req);
7    ...
```

Figure 3. Snippet of invokation code

- *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 invokation. 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 invokation 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 *srvc* is syntactically equal to the initial service; (ii) assume that the initial service does not leak the secret key; and (iii) hence infer that *srvc* 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 φ, then the actions performed by c must also satisfy φ, 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 CONFINE. To explain this rule, we introduce some notation. Let τ denote types in System M that include postconditions for computations and, specifically, let $\texttt{cmp}(\tau, \varphi)$ denote the monadic type of computations that return a value of type τ and whose actions satisfy the predicate φ. (The notation $\texttt{cmp}(\tau, \varphi)$ is simpler than our actual computation types, but it suffices for the explanation here.)

As an illustration of our CONFINE rule, consider any closed expression e. Assume that e does not contain any primitive actions. Then, we claim that for any φ, e has the type $\texttt{cmp}(\texttt{bool}, \varphi) \rightarrow \texttt{cmp}(\texttt{bool}, \varphi)$. To understand this claim, assume that φ is the property "the action is not a write to memory". To show that $e : \texttt{cmp}(\texttt{bool}, \varphi) \rightarrow \texttt{cmp}(\texttt{bool}, \varphi)$, we must show that for any $v : \texttt{cmp}(\texttt{bool}, \varphi)$, $e\,v : \texttt{cmp}(\texttt{bool}, \varphi)$. Hence, we must show that the actions performed by the computation, say c, that $e\,v$ 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 c could write is by invoking the computation v. However, because $v : \texttt{cmp}(\texttt{bool}, \varphi)$, v does not write. Hence, $e\,v : \texttt{cmp}(\texttt{bool}, \varphi)$.

In fact, we can assign e any type, including higher-order function types, as long as the effects in that type are φ. Let the predicate *confine* (τ) (φ) mean that $\varphi = \varphi'$ for all nested types of the form $\texttt{comp}(\tau', \varphi')$ in τ. Let *confine* (Γ) (φ) mean that every type τ that Γ maps to satisfies *confine* (τ) (φ). Let $\texttt{fa}(e) = \emptyset$ 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 τ's nested effects are φ, and the types of the free variables in e also only have φ as effects, then $e : \tau$ with any predicate φ. (Our actual typing rule, shown in Section 4.1 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.)

$$\frac{\texttt{fa}(e) = \emptyset \qquad \texttt{fv}(e) \in \Gamma \qquad \textit{confine}\ (\tau)\ (\varphi) \qquad \textit{confine}\ (\Gamma)\ (\varphi)}{\Gamma \vdash e : \tau}\ \textsc{Confine}$$

In our Memoir example, we use the CONFINE 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 CONFINE rule, we can claim that *srvc*, 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 CONFINE rule we assume that the expression is syntactically free of actions and that all of its free variables are in Γ. 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 *srvc* 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 *srvc* invoked later must be the same program as the intial service. To allow typing *srvc*, 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.

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash e = e'\ \mathsf{true}}{\Gamma \vdash e' : \tau}\ \textsc{Eq}$$

The EQ rule assigns the type τ 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 [10], 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 τ, include type variables (X), a base type b, dependent function types $(\Pi x{:}\tau_1.\tau_2)$, and polymorphic function types $(\forall X.\tau)$. Since System M focuses on deriving trace properties of programs, the difference between base types such as unit and 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 $\texttt{comp}(c)$ is assigned a monadic type $\texttt{comp}(\eta_c)$, where η_c is a closed computation type. A

Expr types	τ	::=	$X \mid \mathtt{b} \mid \Pi x{:}\tau_1.\tau_2 \mid \forall X.\tau \mid \mathtt{comp}(\eta_c) \mid \mathtt{any}$
Comp types	η	::=	$x{:}\tau.\varphi \mid \varphi \mid (x{:}\tau.\varphi, \varphi')$
Closed c types	η_c	::=	$u_1.u_2.i.(x{:}\tau.\varphi_1, \varphi_2)$
		$\mid$	$\Pi x{:}\tau.u_1.u_2.i.(y{:}\tau.\varphi_1, \varphi_2)$
Assertions	φ	::=	$P \mid e_1 = e_2 \mid \varphi\ e \mid \top \mid \bot \mid \neg\varphi$
		$\mid$	$\varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \forall x{:}\tau.\varphi \mid \exists x{:}\tau.\varphi$
Action Kinds	α	::=	$\mathtt{Act}(\eta_c) \mid \Pi x{:}\tau.\alpha \mid \forall X.\alpha$
Type var ctx	Θ	::=	$\cdot \mid \Theta, X$
Signatures	Σ	::=	$\cdot \mid \Sigma, A :: \alpha$
Logic var ctx	Γ^L	::=	$\cdot \mid \Gamma^L, x : \mathtt{b} \mid \Gamma^L, x : \mathtt{any}$
Typing ctx	Γ	::=	$\cdot \mid \Gamma, x : \tau$
Formula ctx	Δ	::=	$\cdot \mid \Delta, \varphi$
Exec ctx	Ξ	::=	$u_b : b, u_e : b, i : b$

Figure 4. Types and typing contexts

closed computation type $u_1.u_2.i.(x{:}\tau.\varphi_1, \varphi_2)$ contains two post-conditions, φ_1 and φ_2. Both are interpreted relative to a trace $\mathcal{T}$. φ_1, the *partial correctness assertion*, holds whenever a computation of this type finishes execution on the trace. It is parametrized by the id i of the thread that runs the computation, the interval $(u_b, u_e]$ during which the computation runs and the return value x of the computation. φ_2, called the *invariant assertion*, holds while a computation of the computation type is still executing (or is stuck), but has not returned. It is parametrized by the id i of the thread running the computation and the time interval $(u_b, u_e]$ over which the computation has executed. Formally, a suspended computation $\mathtt{comp}(c)$ has type $\mathtt{comp}(u_1.u_2.i.(x{:}\tau.\varphi_1, \varphi_2))$ if the following two properties hold for every trace $\mathcal{T}$: (1) if a thread ι on trace $\mathcal{T}$ begins to run c at time U_1 and at time U_2, c returns an expression e, then e has type τ, and $\mathcal{T}$ satisfies $\varphi_1[U_1, U_2, \iota, e/u_1, u_2, i, x]$; (2), if a thread ι on trace $\mathcal{T}$ begins to run c at time U_1 and at time U_2, c has not finished, then $\mathcal{T}$ satisfies $\varphi_2[U_1, U_2, \iota/u_1, u_2, i]$. 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 $\Pi x{:}\tau.u_1.u_2.i.(y{:}\tau.\varphi_1, \varphi_2)$, discussed in more detail with typing rules. If f has this type, then for any $e : \tau$, $(f\ \ e)$ is a recursive computation of closed computation type $u_1.u_2.i.(y{:}\tau.\varphi_1, \varphi_2)[e/x]$.

Assertions, 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 $\mathtt{Act}(\eta_c)$, where η_c 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. Γ^L contains logical variable type bindings. These variables can only be of the type $\mathtt{b}$ or $\mathtt{any}$. Γ contains dependent variable type bindings. Δ contains logical assertions. The ordered context $\Xi = u_b, u_e, i$ provides reference time points and a thread id to typing judgments for computations. When typing a computation, $(u_b, u_e]$ 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.

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

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 $u_b.u_e.i.\varphi$, which we call an invariant. Variables u_b, u_e, and i have the same meaning as the context Ξ, and may appear free in φ. Rules indexed with $u_b.u_e.i.\varphi$ 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 2 and those induced by other rules. We call the former effectful and the latter non-effectful or silent. The typing judgment $\Xi; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash \varphi\ \mathsf{silent}$ 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.

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

The type system may be extended with other sound rules for this judgment. For instance, the following is a trivially sound rule: $u_b.u_e.i; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash (\forall l, t, u_b{<}t \leq u_e \Rightarrow \neg\mathsf{Read}\ i\ l\ t)\ \mathsf{silent}$. If a thread i is not performing any action during time interval $(u_b, u_e]$, then it does not read memory during that time interval.

Partial correctness typing for computations Figure 5 shows selected rules for establishing partial correctness postconditions of computations. The judgment $u_1, u_2, i; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash c : x{:}\tau.\varphi$ means that if in trace $\mathcal{T}$ any thread with id ι begins to execute computation c at time U_1, and at time U_2, c returns an expression e, and $\mathcal{T}$ satisfies all the formulas in Δ, then e has type τ, and $\mathcal{T}$ also satisfies $\varphi[U_1, U_2, \iota, e/u_1, u_2, i, x]$.

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 $a :: \mathtt{Act}(u_1.u_2.i.(x{:}\tau.\varphi_1, \varphi_2))$, 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 $u_2 : \mathtt{b}$ and $i : \mathtt{b}$, because they may appear free in Γ and Δ. For brevity, we elide the types for variables of type $\mathtt{b}$, as they are obvious from the context.

Rule RET assigns e's type to $\mathtt{ret}(e)$. The trace $\mathcal{T}$ containing the evaluation of $\mathtt{ret}(e)$ satisfies two properties, which appear in the postcondition of $\mathtt{ret}(e)$. First, the return expression, which is bound to x, is e (assertion $(x = e)$). Second, $\mathcal{T}$ satisfies any property φ such that $\varphi\ \mathsf{silent}$ holds. This is because reduction of $\mathtt{ret}(e)$ is silent. Here e is typed under the time point u_2, indicating that e can only be evaluated after u_2.

Rule SEQC types the sequential composition $\mathtt{letc}(c_1, x.c_2)$. Starting at time point u_0 and returning at u_3, the execution of $\mathtt{letc}(c_1, x.c_2)$ in any thread i can be divided into three segments for some u_1, u_2: between time u_0 and u_1, where thread i takes only a silent step, pushing $x.c_2$ onto the stack; between time u_1 and u_2, where the computation c_1 runs; and between time u_2 and u_3, where c_2 runs. The first three premises of SEQC assert the effects of each these three segments. When type checking c_2, the facts learned from the execution so far (φ_0 and φ_1) are included in the context. The fourth premise checks that φ is the logical consequence of the conjunction of the three evaluation segments' properties.

Partial correctness typing

$$\frac{\begin{array}{l} u_1;\Theta;\Sigma;\Gamma^L,u_2,i;\Gamma;\Delta \vdash_Q a :: \mathtt{Act}(u_b.u_e.j.(x{:}\tau.\varphi_1,\varphi_2)) \\ u_1,u_2,i;\Theta;\Sigma;\Gamma^L;\Gamma;\Delta \vdash \varphi \ \mathsf{silent} \\ \mathtt{fv}(a) \in \mathtt{dom}(\Gamma) \qquad \text{let } \gamma = [u_1,u_2,i/u_b,u_e,j] \\ d;\Sigma;\Gamma^L;\Gamma \vdash u_1.u_2.i.(x{:}\tau.\varphi_1\gamma,\varphi_2\gamma \wedge \varphi) \ \mathsf{ok} \end{array}}{u_1,u_2,i;\Theta;\Sigma;\Gamma^L;\Gamma;\Delta \vdash_Q \mathtt{act}(a) : (x{:}\tau.\varphi_1\gamma,\varphi_2\gamma \wedge \varphi)} \ \textsc{Act}$$

$$\frac{\begin{array}{l} u_2;\Theta;\Sigma;\Gamma^L,u_1,i;\Gamma;\Delta \vdash_Q e : \tau \\ u_1,u_2,i;\Theta;\Sigma;\Gamma^L;\Gamma;\Delta \vdash \varphi \ \mathsf{silent} \qquad \mathtt{fv}(e) \subseteq \mathtt{dom}(\Gamma) \end{array}}{u_1,u_2,i;\Theta;\Sigma;\Gamma^L;\Gamma;\Delta \vdash_Q \mathtt{ret}(e) : x{:}\tau.((x=e) \wedge \varphi)} \ \textsc{Ret}$$

$$\frac{\begin{array}{l} u_0,u_1,i;\Theta;\Sigma;\Gamma^L;u_3,\Gamma;\Delta,u_0 \le u_1 \vdash \varphi_0 \ \mathsf{silent} \\ u_1,u_2,i;\Theta;\Sigma;\Gamma^L,u_0 : \mathsf{b},u_3;\Gamma;\Delta,u_1 < u_2,\varphi_0 \\ \quad \vdash_Q c_1 : x{:}\tau.\varphi_1 \\ u_2,u_3,i;\Theta;\Sigma;\Gamma^L,u_0,u_1;\Gamma,x:\tau;\Delta,u_2 < u_3,\varphi_0,\varphi_1 \\ \quad \vdash_Q c_2 : y{:}\tau'.\varphi_2 \\ \Theta;\Sigma;\Gamma^L,u_1,u_2,u_0,u_3,i;\Gamma,x{:}\tau,y:\tau';\Delta \\ \quad \vdash (\varphi_0 \wedge \varphi_1 \wedge \varphi_2) \Rightarrow \varphi \ \mathsf{true} \\ \Theta;\Sigma;\Gamma^L,u_0,u_3,i;\Gamma,y:\tau' \vdash \varphi \ \mathsf{ok} \\ \mathtt{fv}(\mathtt{letc}(c_1,x.c_2)) \subseteq \mathtt{dom}(\Gamma) \end{array}}{u_0,u_3,i;\Theta;\Sigma;\Gamma^L;\Gamma;\Delta \vdash_Q \mathtt{letc}(c_1,x.c_2) : y{:}\tau'.\varphi} \ \textsc{SeqC}$$

$$\frac{\begin{array}{l} u_0,u_1,i;\Theta;\Sigma;\Gamma^L,u_3;\cdot;\Delta,u_0 \le u_1 \vdash \varphi_0 \ \mathsf{silent} \\ u_1,u_2,i;\Theta;\Sigma;\Gamma^L,u_0 : \mathsf{b},u_3;\cdot;\varphi_0,u_1 \le u_2 \\ \quad \vdash_Q c_1 : x{:}\tau.\varphi_1 \\ u_2,u_3,i;\Theta;\Sigma;\Gamma^L;u_0,u_1;x:\tau;\Delta,u_2 \le u_3,\varphi_0,\varphi_1 \\ \quad \vdash_{Q2} c_2 : y{:}\tau'.\varphi_2 \\ \Theta;\Sigma;\Gamma^L;u_0,u_3,i;\Gamma,u_1,u_2,y:\tau';\Delta \\ \quad \vdash (\varphi_0 \wedge \varphi_1 \wedge \varphi_2) \Rightarrow \varphi \ \mathsf{true} \\ \Theta;\Sigma;\Gamma^L;u_0,u_3,i,\Gamma,y:\tau' \vdash \varphi \ \mathsf{ok} \end{array}}{u_0,u_3,i;\Theta;\Sigma;\Gamma^L;\Gamma;\Delta \vdash_{Q2} (c_1;c_2) : y{:}\tau'.\varphi} \ \textsc{SeqCComp}$$

Invariant typing

$$\frac{\begin{array}{l} \Theta;\Sigma;\Gamma^L,u_0,u_3,i;\Gamma;\Delta \vdash \varphi \ \mathsf{ok} \\ u_0,u_1,i;\Theta;\Sigma;\Gamma^L,u_3;\Gamma;\Delta,u_0 \le u_1 \vdash \varphi_0 \ \mathsf{silent} \\ u_0,u_3,i;\Theta;\Sigma;\Gamma^L;\Gamma;\Delta,u_0 \le u_3 \vdash \varphi_0' \ \mathsf{silent} \\ u_1,u_2,i;\Theta;\Sigma;\Gamma^L,u_0 : \mathsf{b},u_3;\Gamma;\Delta,u_1 < u_2,\varphi_0 \\ \quad \vdash_Q c_1 : x{:}\tau.\varphi_1 \\ u_1,u_3,i;\Theta;\Sigma;\Gamma^L;\Gamma;\Delta,u_0 : \mathsf{b},u_1 \le u_3,\varphi_0 \vdash_Q c_1 : \varphi_1' \\ u_2,u_3,i;\Theta;\Sigma;\Gamma^L;\Gamma;\Delta,u_0,u_1,x:\tau,u_2 \le u_3,\varphi_0,\varphi_1 \\ \quad \vdash_Q c_2 : \varphi_2 \\ \Theta;\Sigma;\Gamma^L,u_0,u_3,i;\Gamma;\Delta \vdash \varphi_0' \Rightarrow \varphi \ \mathsf{true} \\ \Theta;\Sigma;\Gamma^L,u_0,u_3,i;\Gamma,u_1;\Delta \vdash (\varphi_0 \wedge \varphi_1') \Rightarrow \varphi \ \mathsf{true} \\ \Theta;\Sigma;\Gamma^L,u_0,u_3,i;\Gamma,u_1,u_2,x{:}\tau;\Delta \\ \quad \vdash (\varphi_0 \wedge \varphi_1 \wedge \varphi_2) \Rightarrow \varphi \ \mathsf{true} \\ \mathtt{fv}(\mathtt{letc}(c_1,x.c_2)) \subseteq \mathtt{dom}(\Gamma) \end{array}}{u_0,u_3,i;\Theta;\Sigma;\Gamma^L;\Gamma;\Delta \vdash_Q \mathtt{letc}(c_1,x.c_2) : \varphi} \ \textsc{SeqCI}$$

Figure 5. Selected Rules for Computation Typing

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 c_1 and c_2 must be empty. Because the free variables in c_1 are place holders for expressions that satisfy an invariant φ_1, while the free variables in c_2 are for ones that satisfy a different invariant φ_2, c_1 and c_2 cannot share free variables except those in Γ^L. 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: c_1 executes sandboxed code that satisfies φ_1 and c_2 either contains no sandboxed programs, or ones that satisfy φ_2 .

Invariant typing for computations The meaning of the invariant typing judgment $u_1,u_2,i;\Theta;\Sigma;\Gamma^L;\Gamma;\Delta \vdash c : \varphi$ is the following: Assuming that on a trace $\mathcal{T}$, thread ι 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 $\mathcal{T}$ satisfies assumptions in Δ, then $\mathcal{T}$ also satisfies $\varphi[U_1,U_2,\iota/u_1,u_2,i]$.

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 Σ and the effect of being silent.

Next, we explain the rule SEQCI for the sequencing statement $\mathtt{letc}(c_1,x.c_2)$. We need to consider three cases when deriving the invariant assertion φ of $\mathtt{letc}(c_1,x.c_2)$ in the interval $(u_0,u_3]$: (1) the computation has not started until u_3 (2) the computation c_1 started but has not returned until u_3, (3) the computation c_1 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 φ holds.

For example, $\mathtt{comp}(\mathtt{letc}(\mathtt{act}(\mathsf{read}\ e), x.\mathtt{ret}x))$ can be assigned the following type. Predicate $(\mathsf{mem}\ l\ v\ u)$ is true when at time u, memory location l is allocated and stores the expression v. Predicate $\mathsf{eval}\ e\ e'$ is true if e β-reduces to e', which cannot reduce further. $\mathsf{Write}\ \iota\ l\ e\ u$ states that thread ι writes to address l expression 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.

$$\begin{array}{l} \mathtt{comp}(u_b.u_e.i.(r{:}\mathsf{any}.\forall l,v,\ \mathsf{eval}\ e\ l \wedge \mathsf{mem}\ l\ v\ u_e \Rightarrow y = e, \\ \qquad\qquad \forall l,v,u,u_b < u \le u_e \Rightarrow \neg\mathsf{write}\ i\ l\ v\ u)) \end{array}$$

Fixpoint computation The fixpoint is typed under a time point u, which is the earliest time when the fixpoint is unrolled.

$$\frac{\begin{array}{l} \Gamma_1 = y:\tau, f : \Pi y{:}\tau.\mathtt{comp}(u_1.u_3.i.(x{:}\tau_1.\varphi,\varphi')) \\ u_1,u_2,i;\Theta;\Sigma;\Gamma^L;\Gamma;\Delta,u \le u_1 \le u_2 \vdash \varphi_0 \ \mathsf{silent} \\ u_2,u_3,i;\Theta;\Sigma;\Gamma^L,u_1,u;\Gamma,\Gamma_1;\Delta,u_2 < u_3,\varphi_0 \vdash_Q c : x{:}\tau_1.\varphi_1 \\ u_2,u_3,i;\Theta;\Sigma;\Gamma^L;u_1,u;\Gamma,\Gamma_1;\Delta,u_2 \le u_3,\varphi_0 \vdash_Q c : \varphi_2 \\ \Theta;\Sigma;\Gamma^L,u_1,u,u_2,u_3,i;\Gamma,\Gamma_1,x:\tau_1;\Delta \vdash (\varphi_0 \wedge \varphi_1) \Rightarrow \varphi \ \mathsf{true} \\ \Theta;\Sigma;\Gamma^L,u_1,u_2,u_3,i,u;\Gamma,\Gamma_1;\Delta \vdash (\varphi_0 \wedge \varphi_2 \Rightarrow \varphi') \ \mathsf{true} \\ \Theta;\Sigma;\Gamma^L,u_1,u_3,i,u;\Gamma,y:\tau;\Delta \vdash \varphi_0[u_3/u_2] \Rightarrow \varphi' \ \mathsf{true} \\ \Theta;\Sigma;\Gamma^L,u;\Gamma \vdash \Pi y{:}\tau.u_1.u_3.i.(x{:}\tau_1.\varphi,\varphi') \ \mathsf{ok} \\ \mathtt{fv}(\mathtt{fix}(f(y).c)) \in \mathtt{dom}(\Gamma) \end{array}}{u;\Theta;\Sigma;\Gamma^L;\Gamma;\Delta \vdash_Q \mathtt{fix}(f(y).c) : \Pi y{:}\tau.u_1.u_3.i.(x{:}\tau_1.\varphi,\varphi')} \ \textsc{Fix}$$

Rule FIX simultaneously establishes the partial correctness and invariant assertions of a fixpoint. The third and fourth premises establish the partial correctness and invariant assertions of the body c of the fixpoint. The fifth premise checks that the specified partial correctness assertion φ is entailed by the conjunction of the assertions of a silent thread and the assertion of the body. The next two premises check the invariant assertion φ'. For example, $\mathtt{fix}\ f(x).\mathsf{write}\ x\ 0; \mathsf{read}\ x; \mathtt{lete}(f(x{+}1); z.\mathtt{ret}\ z)$ has the type:

$$\begin{array}{l} \Pi x{:}\mathsf{b}.u_b.u_e.i.(y{:}\mathsf{any}.\bot, \\ \qquad \forall u,l,v,u_b < u \le u_e \wedge \mathsf{read}\ i\ l\ u \\ \qquad\qquad \Rightarrow \exists u', u' < u \wedge \mathsf{write}\ i\ l\ v\ u') \end{array}$$

Expression typing Similar to the fixpoint, the expression typing judgment is parameterized over a time point u, which is the earliest time point that e is evaluated. Recall that the typing rule for $\mathtt{ret}(e)$ types e under the time point when $\mathtt{ret}(e)$ returns. This is because e can only be evaluated after $\mathtt{ret}(e)$ finishes. Most expression typing rules are standard. A representative subset is listed in Figure 6.

$$\frac{\begin{array}{l}u_1, u_2, i; \Theta; \Sigma; \Gamma^L; u_e, \Gamma; \Delta, u_1 \geq u_e \vdash_Q c : (x{:}\tau.\varphi_1, \varphi_2) \\ \Theta; \Sigma; \Gamma^L, u_e{:}\mathtt{b}, u_1{:}\mathtt{b}, u_2{:}\mathtt{b}, i{:}\mathtt{b}; \Gamma, x : \tau; \Delta \vdash \varphi_1 \Rightarrow \varphi_1' \ \mathsf{true} \\ \Theta; \Sigma; \Gamma^L, u_e{:}\mathtt{b}, u_1{:}\mathtt{b}, u_2{:}\mathtt{b}, i{:}\mathtt{b}; \Gamma; \Delta \vdash \varphi_2 \Rightarrow \varphi_2' \ \mathsf{true} \\ \Theta; \Sigma; \Gamma^L, u_e{:}\mathtt{b}; \Gamma \vdash u_1.u_2.i.(x{:}\tau.\varphi_1', \varphi_2') \ \mathsf{ok} \\ \mathtt{fv}(c) \subseteq \mathtt{dom}(\Gamma)\end{array}}{u_e; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_Q \mathtt{comp}(c) : \mathtt{comp}(u_1.u_2.i.(x{:}\tau.\varphi_1', \varphi_2'))}\ \textsc{Comp}$$

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

$$\frac{\begin{array}{l}\varphi \text{ is trace composable} \\ u_b, u_e, i; \Theta; \Sigma; \Gamma^L, u; \Gamma; \Delta \vdash \varphi \ \mathsf{silent} \\ u_b{:}\mathtt{b}, u_e{:}\mathtt{b}, i{:}\mathtt{b} \vdash \varphi \ \mathsf{ok} \qquad \mathtt{fa}(e) = \emptyset \qquad \mathtt{fv}(e) \subseteq \Gamma \\ \mathit{confine}\ (\tau)\ (u_b.u_e.i.\varphi) \qquad \mathit{confine}\ (\Gamma)\ (u_b.u_e.i.\varphi)\end{array}}{u; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_{u_b.u_e.i.\varphi} e : \tau}\ \textsc{Confine}$$

$$\frac{u; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash e : \tau \qquad u_b{:}\mathtt{b}, u_e{:}\mathtt{b}, i{:}\mathtt{b} \vdash \varphi \ \mathsf{ok}}{u; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_{u_b.u_e.i.\varphi} e : \tau}\ \textsc{Conf-sub}$$

Figure 6. Selected expression typing rules

Rule COMP assigns a monadic type to a suspended computation by checking the computation. Since the suspended computation can only execute after u_e, the logical context of the first premise can safely assume that the beginning time point of c is no earlier than u_e. As usual, the rule also builds-in weakening of postconditions.

The rule EQ, motivated in Section 3.1, assigns an expression e', the type of e, if e is syntactically equal to e'.

The rule CONFINE, motivated in Section 3.1, allows us to type an expression from the knowledge that it contains no actions and that its free variables will be substituted with expressions with effect φ. The main generalization from the simpler rule presented in Section 3.1 is that now φ 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 c 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 φ while they execute, then as c executes, it maintains φ.

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

$$\frac{}{\mathit{confine}\ (b)\ (u_b.u_e.i.\varphi)}$$

$$\frac{\mathit{confine}\ (\tau_1)\ (u_b.u_e.i.\varphi) \quad \mathit{confine}\ (\tau_2)\ (u_b.u_e.i.\varphi)}{\mathit{confine}\ (\Pi\_{:}\tau_1.\tau_2)\ (u_b.u_e.i.\varphi)}$$

$$\frac{\mathit{confine}\ (\tau)\ (u_b.u_e.i.\varphi)}{\mathit{confine}\ (\mathtt{comp}(u_b.u_e.i.(x{:}\tau.\varphi, \varphi)))\ (u_b.u_e.i.\varphi)}$$

The CONFINE rule itself does not stipulate any conditions on the predicate φ, other than requiring that φ 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 φ 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 $u_b.u_e.i.\varphi$. This is sound because the first premise does not require the substitutions for Γ to satisfy any specific invariant, so they can be narrowed down to any invariant. The conclusion must be tagged with the invariant φ, because: (1) τ could be a base type, in which case, the invariant is not evident in e's type; and (2) the types in Γ are allowed to contain nested effects that are not φ. 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 $\mathtt{letc}(a_1; z.\mathtt{ret}(\mathtt{comp}(a_2)))$ can be the following, assuming that the effect of action a_1 is $\mathsf{A}_1\ i\ u$, and a_2 is $\mathsf{A}_2\ i\ u$.
$\mathtt{comp}(u_b.u_e.i.(r{:}\ \mathtt{b}.\exists\ u, u_b{<}u{\leq}u_e \wedge \mathsf{A}_2\ i\ u \wedge \exists j, u', u'{<}u \wedge \mathsf{A}_1\ j\ u', \top))$.

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.

$$\frac{\begin{array}{c}u_1, u_2, i; \Theta; \Sigma; \Gamma^L; \cdot; \Delta \vdash c : \varphi \\ \Theta; \Sigma; \Gamma^L; \cdot; \Delta \vdash \mathsf{start}(I, c, u)\ \mathsf{true} \\ \Theta; \Sigma \vdash \Gamma^L, \Gamma\ \mathsf{ok}\end{array}}{\Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash \forall u'{:}\mathtt{b}.(u'{>}u) \Rightarrow \varphi[u, u', I/u_1, u_2, i]\ \mathsf{true}}\ \textsc{Honest}$$

If we know that a thread ι starts executing at time u with payload computation c (premise $\mathsf{start}(\iota, c, u)$) and computation c has an invariant postcondition φ, then we can conclude that at any later point u', φ holds for the interval $(u, u']$. The condition that c be typed under an empty Γ 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_i with the key $skey$, whenever we invoke the service with $state$ at a time point u, later than u_i, it must be the case that, the service was either initialized or produced the state $state$ at a time point u'. Moreover, there is no invokations of the service between u' and u.

$$\begin{array}{l}\forall u_i, state, state', skey, i_{init}, s_{init} \\ \quad \mathsf{service\_init}(i_{init}, skey, service, s_{init})@u_i \Rightarrow \\ \quad\quad \forall u > u_i.\ \mathsf{service\_invoke}(i, skey, state, state')@u \Rightarrow \\ \quad\quad\quad \exists j, u' < u.\ ((\exists s.\mathsf{service\_invoke}(j, skey, s, state)@u' \\ \quad\quad\quad\quad\quad \vee \mathsf{service\_try}(j, skey, state)@u' \\ \quad\quad\quad\quad\quad \vee \mathsf{service\_init}(j, skey, state)@u') \\ \quad\quad\quad\quad \wedge (\forall j'.\ \neg\mathsf{service\_invoke}(j', skey, \cdots) \circ (u', u)]))\end{array}$$

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 $s\_hash = H(h||code\_hash(service))$. Here, the term $H(x)$ denotes hash of x, $||$ denotes concatenation, h is any value and $code\_hash(x)$ 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 s_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 $srvc$, we need to type $srvc$. Recall that $srvc$ 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 $srvc$ by typing against the TPM interface. The particular invariant type we wish to derive about $srvc$ 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 $srvc$ 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).

$$(srvc\ ExtendPCR\ ResetPCR\ \cdots)\ (state, req):$$
$$\mathsf{cmp}(u_b, u_e, i.\ \neg\mathsf{PCRPrefix}(\mathtt{pcr17}, s\_hash)@u_b \Rightarrow$$
$$\forall u \in (u_b, u_e].\ (\mathsf{InLLSession}(u, runmodule, i)$$
$$\Rightarrow \neg\mathsf{PCRPrefix}(\mathtt{pcr17}, s\_hash)@u)$$

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 $ExtendPCR$ 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 $runmodule$ to read the NVRAM location $Nloc$ (i.e., the first invariant of $srvc$ in Section 3.1).

Similarly, we can prove that the permissions on $Nloc$ are always tied to PCR 17 being s_hash, by typing $srvc$ 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 $srvc$, the state of PCR 17 must be $H(h||code\_hash(srvc))$ for some h. Therefore, by $Nloc$'s access control mechanism, we prove that $H(h||code\_hash(srvc)) = s\_hash$ and therefore $srvc = service$ (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 $srvc$. Specifically, we assume that $service$ has the following type τ_{sec} (which means that if the input of $service$ does not contain a secret s then the output doesn't contain it) and an invariant $\mathsf{KeepsSecret}(i, s, Nloc)$ (which means that s is not sent out on the network and the only NVRAM location s possibly written to is $Nloc$).

$$\tau_{sec} = \Pi i : \mathtt{msg}.\ \mathsf{cmp}(u_b, u_e, i.$$
$$(x : \mathtt{msg}.\forall s.\ \neg\mathsf{Contains}(i, s) \Rightarrow \neg\mathsf{Contains}(x, s),$$
$$\forall s.\ \neg\mathsf{Contains}(i, s) \Rightarrow \mathsf{KeepsSecret}(i, s, Nloc) \circ (u_b, u_e]))$$

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

5. Semantics and Soundness

We build a step-indexed semantic model [2] 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 $u_b.u_e.i.\varphi$ ($\mathcal{RE}_{INV}[\![u_b.u_e.i.\varphi]\!]$), and the other is an invariant-indexed semantics for types ($\mathcal{RE}(u_b.u_e.i.\varphi)[\![\tau]\!]$). These two sets coincide when *confine* (τ) $(u_b.u_e.i.\varphi)$ holds (Lemma 1).

5.1 A Step-indexed Semantics for Invariants

We define $\mathcal{RV}_{INV}[\![\Phi]\!]_{\mathcal{T};u}$, $\mathcal{RE}_{INV}[\![\Phi]\!]_{\mathcal{T};u}$, $\mathcal{RC}_{INV}[\![\Phi]\!]_{\mathcal{T};u}$ ($\Phi = u_b.u_e.i.\varphi$), the sets of step-indexed normal forms, expressions, and computations that satisfy the invariant φ respectively. $\mathcal{T}$ 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 φ below. An indexed computation (k, c) belongs to this relation if the following holds: (1) during any interval u_B and u_E when thread ι executes c on $\mathcal{T}$, $\varphi[u_B, u_E, \iota/u_b, u_e, i]$ holds on $\mathcal{T}$ 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.

$\mathcal{RC}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u} =$
$\{(k, c) \mid \forall u_B, u_E, \iota, u \le u_B \le u_E,$
let $\gamma = [u_B, u_E, \iota/u_1, u_2, i]$,
j_b is the length of the trace from time u_B to the end of $\mathcal{T}$
j_e is the length of the trace from time u_E to the end of $\mathcal{T}$
$k \ge j_b > j_e$,
the configuration at time u_1 is $\xrightarrow{u_B} \sigma_b \triangleright \cdots, \langle \iota; x.c' :: K; c\rangle \cdots$
the configuration at time u_E is $\xrightarrow{u_E} \sigma_e \triangleright \cdots, \langle \iota; K; c'[e'/x]\rangle \cdots$
between u_B and u_E, the stack of thread i always contains $x.c'::K$
$\Longrightarrow (j_e, e') \in \mathcal{RE}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u_E}$ and $\mathcal{T} \vDash_\theta \varphi[e'/x]\}$
$\cap \{(k, c) \mid \forall u_B, u_E, \iota, u \le u_B \le u_E$, let $\gamma = [u_B, u_E, \iota/u_1, u_2, i]$,
j_b is the length of the trace from time u_B to the end of $\mathcal{T}$,
j_e is the length of the trace from time u_E to the end of $\mathcal{T}$
$k \ge j_b \ge j_e$,
the configuration at time u_B is $\xrightarrow{u_B} \sigma_b \triangleright \cdots, \langle \iota; x.c' :: K; c\rangle \cdots$
between u_B and u_E (inclusive), the stack of thread i always
contains prefix $x.c'::K$
$\Longrightarrow \mathcal{T} \vDash_\theta \varphi\}$

We explain some parts of the definition. At time u_B, thread ι begins to run c, which is formalized by requiring that the thread $\langle \iota; K; c\rangle$ is in the configuration right after time u_B. At time u_E, c returns an expression e' to its context, which is formalized by requiring that thread ι's top frame is popped off the stack with e' substituted for x, and that the top frame remains unchanged between u_B and u_E. Both u_B and u_E are within the last k configurations of the trace because the length of the trace is n and $k \ge j_b > j_e$. The earliest time point to interpret e' is u_E, which is when e' is returned. The index for the returned expression e' is j_e, which is less than k. Hence, our step-indices count the number of remaining steps in the trace. Moreover, these remaining steps include not just steps of the thread containing c, but also other threads. This ensures the computation c's postconditions hold even when it executes concurrently with other threads (robust safety; Theorem 4). For the second set, c must not have finished at u_E, so between u_b and u_e, no frame on the stack $x.c' :: K$ should have been popped.

The relation $\mathcal{RV}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}$ includes all normal expressions that are not introduction forms (i.e. functions and suspended computations). These normal forms cannot be further reduced in any evaluation context, and therefore do not have any effects (they are silent). A function is in this relation if, given arguments maintaining the same invariant, the function body also maintains that invariant. As is standard, the step-index of the argument is smaller than that of the function because function application consumes a step. The case of polymorphic functions is defined similarly. A sus-

pended computation $\texttt{comp}(c)$ belongs to this relation if c belongs to the $\mathcal{RC}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}$ relation defined earlier.

$\mathcal{RV}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u} = \{(k, \texttt{nf}) \mid \texttt{nf} \neq \lambda x.e, \Lambda X.e, \texttt{comp}(c)\}$
$\cup\{(k, \texttt{comp}(c)) \mid (k, c) \in \mathcal{RC}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}\}$
$\cup\{(k, \lambda x.e') \mid \forall j, u', j < k, u' \geq u$
$\quad (j, e') \in \mathcal{RE}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u'}$
$\quad \Longrightarrow (j, e[e'/x]) \in \mathcal{RE}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u'}\}$
$\cup\{(k, \Lambda x.e) \mid \forall j, j < k \Longrightarrow (j, e) \in \mathcal{RE}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}\}$

The definition of the $\mathcal{RE}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}$ relation is standard: if e evaluates to a normal form $\texttt{nf}$ in m steps, then $\texttt{nf}$ has to be in the value relation indexed by the number of the remaining steps.

$\mathcal{RE}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}=$
$\{(k, e) \mid \forall 0 \leq m \leq k, e \to^m e' \not\to$
$\quad \Longrightarrow (n - m, e') \in \mathcal{RV}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}\}$

This relation includes all programs (including ill-typed ones) that satisfy the invariant if executed in a context that satisfies that invariant. This relation justifies the soundness of CONFINE rule. Confined adversary-supplied code is in the $\mathcal{RE}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}$ relation (Lemma 2).

5.2 A Step-indexed Model for Types

As programs include adversarial code, which requires its evaluation context to maintain an invariant, the semantics of types need to be indexed by invariants of the form $u_b.u_e.i.\varphi$.

Types The interpretation of an expression type τ is a semantic type, written C. Each C is a set of pairs; each pair contains a step-index and an expression. The expression has to be in normal form, denoted $\texttt{nf}$, that cannot be reduced further under call-by-name β-reduction. C contains the set of all possible indices and all syntactically well-formed normal forms. This is used to interpret the type $\texttt{any}$ of untyped programs. As usual, we require that C be closed under reduction of step-indices. Let $\mathcal{P}(S)$ denote the powerset of S. The set of all semantic types is denoted Type.

$\mathsf{Type} \stackrel{def}{=} \{\mathsf{C} \mid \mathsf{C} \in \mathcal{P}(\{(j, \texttt{nf}) \mid j \in \mathbb{N}\}) \wedge$
$\quad (\forall k, \texttt{nf}, (k, \texttt{nf}) \in \mathsf{C} \wedge j < k \Longrightarrow (j, \texttt{nf}) \in \mathsf{C}) \wedge$
$\quad (\forall k, \texttt{nf}, \texttt{nf} \neq \lambda x.e, \Lambda X.e, \texttt{comp}(e) \Longrightarrow (j, \texttt{nf}) \in \mathsf{C})\}$

Interpretation of expression types We define the *value and expression interpretations* of expression types τ (written $\mathcal{RV}(\Phi)[\![\tau]\!]_{\theta;\mathcal{T};u}$ and $\mathcal{RE}(\Phi)[\![\tau]\!]_{\theta;\mathcal{T};u}$), as well as the *interpretation* of computation types η (written $\mathcal{RC}(\Phi)[\![\eta]\!]_{\theta;\mathcal{T};u}$) simultaneously by induction on types ($\Phi = u_b.u_e.i.\varphi$). Let θ denote a partial map from type variables to Type, $\mathcal{T}$ denote the trace that expressions are evaluated on, and u denote the time point after which expressions are evaluated. Figure 11 defines the value and expression interpretations. We omit the cases for $\texttt{any}$ and X.

The interpretation of the base type $\texttt{b}$ is the same as $\mathcal{RV}_{INV}[\![\Phi]\!]_{\theta;\mathcal{T};u}$. The type $\texttt{b}$ itself doesn't specify any effects, and, therefore, expressions in the interpretation of $\texttt{b}$ only need to satisfy the invariant Φ. The interpretation of the function type $\Pi x{:}\tau_1.\tau_2$ is nonstandard: the substitution for the variable x is an expression, not a value. This simplifies the proof of soundness of function application: since System M uses call-by-name β-reduction, the reduction of $e_1\ e_2$ 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 λ abstractions that are in the $\mathcal{RV}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\theta;\mathcal{T};u}$ relation. These are adversary-supplied untyped code, which is required by our type system to satisfy the invariant $u_b.u_e.i.\varphi$.

The interpretation of the monadic type includes suspended computations $(k, \texttt{comp}(c))$ such that (k, c) belongs to the interpretation of computation types, defined below. Because c executes after time 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 $u_b.u_e.i.\varphi$. The interpretation of the $\texttt{any}$ type contains all normal forms.

We lift the value interpretation $\mathcal{RV}(\Phi)[\![\tau]\!]_{\theta;\mathcal{T};u}$ to the expression interpretation $\mathcal{RE}(\Phi)[\![\tau]\!]_{\theta;\mathcal{T};u}$ in a standard way.

Interpretation of formulas Formulas are interpreted on traces. We write $\mathcal{T} \vDash \varphi$ to mean that φ is true on trace $\mathcal{T}$.

$\mathcal{T} \vDash P\ \vec{e}$	iff	$P\ \vec{e} \in \varepsilon(\mathcal{T})$
$\mathcal{T} \vDash \mathsf{start}(I, c, U)$	iff	thread I has c as the active computation with an empty stack at time U on $\mathcal{T}$
$\mathcal{T} \vDash \forall x{:}\tau.\varphi$	iff	$\forall e, e \in [\![\tau]\!]$ implies $\mathcal{T} \vDash \varphi[e/x]$

We assume a valuation function $\varepsilon(\mathcal{T})$ that returns the set of atomic formulas that are true on the trace $\mathcal{T}$. For first-order quantification, we select terms in the denotation of the types ($[\![\tau]\!]$), which is defined as follows:

$[\![\texttt{any}]\!] = \{e \mid e \text{ is an expression}\}$
$[\![\texttt{b}]\!] = \{e \mid e \to^* bv\}$
$[\![\Pi x{:}\tau_1.\tau_2]\!] = \{\lambda x.e \mid \forall e', e' \in [\![\tau_1]\!] \Longrightarrow e_1[e'/x] \in [\![\tau_2]\!]\}$

The types of the logical variables can only be $\texttt{b}$, $\texttt{any}$ 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, $\mathcal{RC}(u_b.u_e.i.\varphi_1)[\![x{:}\tau.\varphi]\!]_{\theta;\mathcal{T};\Xi}$, is a set of step-indexed computations. The trace $\mathcal{T}$ contains the execution of the computation. $\Xi = u_b, u_e, i$ has its usual meaning, except that u_b, u_e, and i are concrete values, not variables.

We define the semantics of the partial correctness type, denoted $\mathcal{RC}(u_b.u_e.i.\varphi_1)[\![x{:}\tau.\varphi]\!]_{\theta;\mathcal{T};\Xi}$, below. Informally, it contains the set of indexed computations c, if $\mathcal{T}$ contains a complete execution of the computation c in the time interval $(u_b, u_e]$ in thread ι such that c returns e' at time u_e and it is also the case that $\mathcal{T}$ satisfies $\varphi[e'/x]$ and that e' has type τ semantically. Similar to the $\mathcal{RC}_{INV}[\![\Phi]\!]_{\mathcal{T};u}$ relation, these remaining steps include not just steps of the thread executing c, but also other threads. The invariant $u_b.u_e.i.\varphi_1$ is used in the specification of the return value.

$\mathcal{RC}(u_b.u_e.i.\varphi_1)[\![x{:}\tau.\varphi]\!]_{\theta;\mathcal{T};u_1,u_2,i} = \{(k, c) \mid$
j_b is the length of the trace from time u_1 to the end of $\mathcal{T}$
j_e is the length of the trace from time u_2 to the end of $\mathcal{T}$
$k \geq j_b > j_e$,
the configuration at time u_1 is $\xrightarrow{u_1} \sigma_b \triangleright \cdots, \langle \iota; x.c' :: K; c\rangle \cdots$
the configuration at time u_2 is $\xrightarrow{u_2} \sigma_e \triangleright \cdots, \langle \iota; K; c'[e'/x]\rangle \cdots$
between u_1 and u_2, the stack of thread i always contains $x.c'{::}K$
$\Longrightarrow (j_e, e') \in \mathcal{RE}(u_b.u_e.i.\varphi_1)[\![\tau]\!]_{\theta;\mathcal{T};u_2}$
and $\mathcal{T} \vDash \varphi[e'/x]\}$

The interpretation for the invariant assertions is defined similarly, and we omit its definition. Because c is being evaluated and produces no return value, the interpretation need not be indexed by an invariant. We write $\_$ in place of the invariant.

5.3 Examples

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

$$\mathsf{next}(\sigma, \mathsf{read}\ e_1\ e_2) = \begin{cases} (\sigma, \sigma(\ell)) & \ell \in \texttt{dom}(\sigma) \\ (\sigma, \mathsf{stuck}) & \ell \notin \texttt{dom}(\sigma) \end{cases}$$

Predicate $\mathsf{stuck}\ \iota\ u$ is true when thread ι 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,

$$
\begin{array}{l}
\mathcal{RV}(u_b.u_e.i.\varphi)[\![\mathtt{b}]\!]_{\theta;\mathcal{T};u} = \{(k,e) \mid (k,e) \in \mathcal{RV}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\theta;\mathcal{T};u}\} \\
\mathcal{RV}(u_b.u_e.i.\varphi)[\![\Pi x{:}\tau_1.\tau_2]\!]_{\theta;\mathcal{T};u} = \{(k,\lambda x.e) \mid \forall j < k, \forall u', u' \geq u, \forall e', (j,e') \in \mathcal{RE}(u_b.u_e.i.\varphi)[\![\tau_1]\!]_{\theta;\mathcal{T};u'} \\
\qquad\qquad \Longrightarrow (j, e_1[e'/x]) \in \mathcal{RE}(u_b.u_e.i.\varphi)[\![\tau_2[e'/x]]\!]_{\theta;\mathcal{T};u'}\} \cup \\
\qquad\qquad \{(k,\mathtt{nf}) \mid \mathtt{nf} \neq \lambda x.e \Longrightarrow (k,\mathtt{nf}) \in \mathcal{RE}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}\} \\
\mathcal{RV}(u_b.u_e.i.\varphi)[\![\forall X.\tau]\!]_{\theta;\mathcal{T};u} = \{(k,\Lambda X) \mid \forall j < k, \forall \mathsf{C} \in \mathsf{Type} \Longrightarrow (j,e') \in \mathcal{RE}(u_b.u_e.i.\varphi)[\![\tau]\!]_{\theta[X \mapsto \mathsf{C}];\mathcal{T};u}\} \cup \\
\qquad\qquad \{(k,\mathtt{nf}) \mid \mathtt{nf} \neq \Lambda X.e \Longrightarrow (k,\mathtt{nf}) \in \mathcal{RE}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}\} \\
\mathcal{RV}(u_b.u_e.i.\varphi)[\![\mathtt{comp}(u_1.u_2.i.(x{:}\tau.\varphi_1,\varphi_2))]\!]_{\theta;\mathcal{T};u} = \\
\quad \{(k,\mathtt{comp}(c)) \mid \forall u_B, u_E, \iota, u \leq u_B \leq u_E, \mathsf{let}\ \gamma = [u_B, u_E, \iota/u_1, u_2, i] \\
\qquad\qquad (k,c) \in \mathcal{RC}(u_b.u_e.i.\varphi)[\![x{:}\tau\gamma.\varphi_1\gamma]\!]_{\theta;\mathcal{T};u_B,u_E,\iota} \cap \mathcal{RC}(\_)[\![\varphi_2\gamma]\!]_{\theta;\mathcal{T};u_B,u_E,\iota}\} \cup \\
\quad \{(k,\mathtt{nf}) \mid \mathtt{nf} \neq \mathtt{comp}(c) \Longrightarrow (k,\mathtt{nf}) \in \mathcal{RE}_{INV}[\![u_1.u_2.i.\varphi]\!]_{\mathcal{T};u}\}
\end{array}
$$

$$
\mathcal{RE}(u_b.u_e.i.\varphi)[\![\tau]\!]_{\theta;\mathcal{T};u} = \{(k,e) \mid \forall j < m, e \rightarrow_\beta^m e' \not\rightarrow \Longrightarrow (k-m,e') \in \mathcal{RV}(u_b.u_e.i.\varphi)[\![\tau]\!]_{\theta;\mathcal{T};u}\}
$$

Figure 7. Semantics for inv-indexed types

the thread does not get stuck and the expression y returned by 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. $(n, \mathtt{act}(\mathsf{read}\ e)) \in$
   $\mathcal{RC}(\Phi)[\![y{:}\mathsf{any}.\forall l, v, \mathsf{mem}\ l\ v\ u_2 \wedge \mathsf{eval}\ e\ l \Rightarrow (y = e) \wedge \neg\mathsf{stuck}\ i@(u_1,u_2)]\!]_{\theta;\mathcal{T};u_1,u_2,i}$
2. $\mathcal{RC}(\Phi)[\![\forall j, l, e, t.(\neg\mathsf{Write}\ j\ l\ e\ t)]\!]_{\theta;\mathcal{T};u_1,u_2,i} = \emptyset$

The second example states that the interpretation of the invariant computation type $(\forall j, l, e, t.(\neg\mathsf{Write}\ j\ l\ e\ t))$, which states that no 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 5.2. We start by defining valid substitutions for contexts. We write $\mathcal{RT}[\![\Theta]\!]$ to denote the set of valid semantic substitutions for Θ. We write $\mathcal{RG}(\Phi)[\![\Gamma]\!]_{\theta;\mathcal{T};u}$ to denote a set of substitutions for variables in Γ. Each indexed substitution is a pair of an index and a substitution γ for variables.

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

Lemma 1 (Indexed types are confined)**.** *confine* (τ) $(u_b.u_e.i.\varphi)$ *implies* $\mathcal{RE}(u_b.u_e.i.\varphi)[\![\tau]\!]_{\theta;\mathcal{T};u} = \mathcal{RE}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}$.

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 e.

Lemma 2 (Invariant confinement)**.** *If* φ *is composable, and thread* ι *silent between time* u_B *and* u_E *implies* $\mathcal{T} \vDash \varphi[u_B, u_E, I/u_b, u_e, i]$*, then* $\mathtt{fa}(e) = \emptyset$*,* $\mathtt{fv}(e) \in \mathtt{dom}(\gamma)$*, and* $(n,\gamma) \in \mathcal{RE}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}$ *imply* $(n, e\gamma) \in \mathcal{RE}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}$.

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 Φ, if the substitution for Γ belongs to the interpretation of types, then the expression (computation) belongs to the interpretation of its type, indexed by the same invariant Φ. For judgments qualified by a specific invariant Φ, the soundness theorem statements are also specific to that Φ.

Theorem 3 (Soundness)**.**

Assume that $\forall A :: \alpha \in \Sigma, \forall \Phi, \mathcal{T}, n, u, (n,A) \in \mathcal{RA}(\Phi)[\![\alpha]\!]_{\cdot;\mathcal{T};u}$,

1. *(a)* $\mathcal{E} :: u : \mathtt{b}; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_\Phi e : \tau$, $\forall \theta \in \mathcal{RT}[\![\Theta]\!]$, $\forall \gamma^L \in [\![\Gamma^L]\!]$, $\forall U, U', U' \geq U$, *let* $\gamma_u = [U/u]$, $\forall \mathcal{T}$, $\forall n, \gamma$, $(n;\gamma) \in \mathcal{RG}(\Phi)[\![\Gamma\gamma_u\gamma^L]\!]_{\theta;\mathcal{T};U'}$, $\mathcal{T} \vDash \Delta\gamma\gamma_u\gamma^L$ *implies* $(n; e\gamma) \in \mathcal{RE}(\Phi)[\![\tau\gamma\gamma_u\gamma^L]\!]_{\theta;\mathcal{T};U'}$
   (b) $\mathcal{E} :: u_1, u_2, i; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_\Phi c : \eta$, $\forall\ u, u_B, u_E$, ι *s.t.* $u \leq u_B \leq u_E$, *let* $\gamma_1 = [u_B, u_E, \iota/u_1, u_2, i]$ $\forall \theta \in \mathcal{RT}[\![\Theta]\!]$, $\forall \gamma^L \in [\![\Gamma^L]\!]$, $\forall \mathcal{T}$, $\forall n, \gamma, (n;\gamma) \in \mathcal{RG}(\Phi)[\![\Gamma\gamma_1\gamma^L]\!]_{\theta;\mathcal{T};u}$, $\mathcal{T} \vDash \Delta\gamma\gamma_1\gamma^L$ *implies* $(n; c\gamma) \in \mathcal{RC}(\Phi)[\![\eta\gamma\gamma_1\gamma^L]\!]_{\theta;\mathcal{T};u_B,u_E,\iota}$
2. *(a)* $\mathcal{E} :: u : \mathtt{b}; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash e : \tau$, $\forall \theta \in \mathcal{RT}[\![\Theta]\!]$, $\forall \gamma^L \in [\![\Gamma^L]\!]$, $\forall U, U', U' \geq U$, *let* $\gamma_u = [U/u]$, $\forall \mathcal{T}$, $\forall \Phi$, $\forall n, \gamma$, $(n;\gamma) \in \mathcal{RG}(\Phi)[\![\Gamma\gamma_u\gamma^L]\!]_{\theta;\mathcal{T};U'}$, $\mathcal{T} \vDash \Delta\gamma\gamma_u\gamma^L$ *implies* $(n; e\gamma) \in \mathcal{RE}(\Phi)[\![\tau\gamma\gamma_u\gamma^L]\!]_{\theta;\mathcal{T};U'}$
   (b) $\mathcal{E} :: u_1, u_2, i; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash c : \eta$, $\forall\ u, u_B, u_E$, ι *s.t.* $u \leq u_B \leq u_E$, *let* $\gamma_1 = [u_B, u_E, \iota/u_1, u_2, i]$ $\forall \theta \in \mathcal{RT}[\![\Theta]\!]$, $\forall \gamma^L \in [\![\Gamma^L]\!]$, $\forall \mathcal{T}$, $\forall \Phi$, $\forall n, \gamma, (n;\gamma) \in \mathcal{RG}(\Phi)[\![\Gamma\gamma_1\gamma^L]\!]_{\theta;\mathcal{T};u}$, $\mathcal{T} \vDash \Delta\gamma\gamma_1\gamma^L$ *implies* $(n; c\gamma) \in \mathcal{RC}(\Phi)[\![\eta\gamma\gamma_1\gamma^L]\!]_{\theta;\mathcal{T};u_B,u_E,\iota}$
   (c) $\mathcal{E} :: \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash \varphi\ \mathsf{true}$, $\forall \theta \in \mathcal{RT}[\![\Theta]\!]$, $\forall \gamma^L \in [\![\Gamma^L]\!]$, $\forall \mathcal{T}$, $\forall \Phi$, $\forall n, \gamma, u, (n;\gamma) \in \mathcal{RG}(\Phi)[\![\Gamma\gamma^L]\!]_{\theta;\mathcal{T};u}$, $\mathcal{T} \vDash \Delta\gamma^L\gamma$ *implies* $\mathcal{T} \vDash \varphi\gamma^L\gamma$

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.(a)*) first uses Lemma 1 to show that a substitution γ for Γ, where γ maps each variable in Γ 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 τ. 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 φ.

To prove the soundness of HONEST, we need to show that given any substitution (n,γ) for Γ, 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,γ) for Γ. Because Γ is empty, we complete the proof by using the induction hypothesis on the first premise given an empty substitution $(m,\cdot)$.

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 $\mathcal{RC}(u_b.u_e.i.\varphi)[\![\eta]\!]_{\theta;\Delta;\Xi}$. A similar theorem holds for partial correctness assertions.

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

- $u_1, u_2, i; \Delta \vdash c : \varphi$, $\mathcal{T} \vDash \Delta$,
- $\mathcal{T}$ *is a trace obtained by executing the parallel composition of threads of ID (*ι_1, .. ι_k*),*
- *at time* U_b*, the computation that thread* ι_j *is about to run is* c
- *at time* U_e*,* c *has not returned*

then $\mathcal{T} \vDash \varphi[U_b, U_e, \iota_j/u_1, u_2, i]$.

6. Discussion

Proving non-stuckness We can use System M's invariant assertions to verify that a program always remains non-stuck. Recall the example from Section 5.3. We can prove non-stuckness for a computation c by showing that it has the invariant postcondition $(\neg\mathsf{stuck}\ i)@(u_b, u_e]$. 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 5.3 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 β-reduction for expressions, which simplies the operational semantics as well as the soundness proofs. Other evaluation strategies we have considered force us to use β-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 β-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 [2].

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^2 and PCL System M is inspired by and based upon a prior program logic, LS^2, for reasoning about safety properties of first-order order programs in the presence of adversaries [14]. The main conceptual difference from LS^2 is that in System M trusted and untrusted components may exchange code and data, whereas in LS^2 this interface is limited to data. Our 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^2. 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^2 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 [23], 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^2. 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 $\mathcal{RC}(\Phi)[\![\eta]\!]_{\theta;\mathcal{T};\Xi}$ 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, Theorems 2 & 4]. In contrast to RCF, we designed System M for verification of general safety properties (so the measurement integrity property can be expressed and verified in System M), but we have not considered automation for System M so far.

F* [31] extends F7 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 constructs proofs present in the type cast. The soundness of type cast in CCA relies on the fact that untrusted code cannot make any assertions and that it can only use those made by trusted code. Similar to CCA, System M also assigns untrusted code descriptive types. CCA checks those types at runtime; whereas the 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 [7, 10–12, 16]. Gurgens et al. uses automaton to model the transitions of TPM APIs [16]. Several results [7, 11, 12] use 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 protocol states and works with a monotonic set of facts. Special techniques need to be applied to use Proverif to analyze stateful protocols such as ones that use TPM PCRs [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+ [19] 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 logical 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 *Cryptology 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. Delaune, S. Kremer, M. D. Ryan, and G. Steel. A formal analysis of authentication in the TPM. In *Proc. FAST'10*, 2011.

[12] S. Delaune, 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. Gürgens, C. Rudolph, D. Scheuermann, M. Atts, and R. Plaga. Security evaluation of scenarios based on the TCG's TPM specification. In *Proc. ESORICS'07*, 2007.

[17] C. B. Jones. Tentative steps toward a development method for interfering 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. Govereau, 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 D. 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.

[31] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *Proc. ICFP*, 2011.

[32] TrustedComputingGroup. TPM library specification. `http://www.trustedcomputinggroup.org/resources/tpm_library_specification`.

A. Term Language and Operational Semantics

Syntax

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

Expr types	τ	::=	$X \mid \mathtt{b} \mid \Pi x{:}\tau_1.\tau_2 \mid \forall X.\tau \mid \mathtt{comp}(\eta_c) \mid \mathtt{any}$
Comp types	η	::=	$x{:}\tau.\varphi \mid \varphi \mid (x{:}\tau.\varphi, \varphi')$
Closed c types	η_c	::=	$u_1.u_2.i.(x{:}\tau.\varphi_1, \varphi_2)$
		$\mid$	$\Pi x{:}\tau.u_1.u_2.i.(y{:}\tau.\varphi_1, \varphi_2)$
Assertions	φ	::=	$P \mid e_1 \equiv e_2 \mid \varphi\ e \mid \top \mid \bot \mid \neg\varphi$
		$\mid$	$\varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \forall x{:}\tau.\varphi \mid \exists x{:}\tau.\varphi$
Action Kinds	α	::=	$\mathtt{Act}(\eta_c) \mid \Pi x{:}\tau.\alpha \mid \forall X.\alpha$
Type var ctx	Θ	::=	$\cdot \mid \Theta, X$
Signatures	Σ	::=	$\cdot \mid \Sigma, A :: \alpha$
Logic var ctx	Γ^L	::=	$\cdot \mid \Gamma^L, x : \mathtt{b} \mid \Gamma^L, x : \mathtt{any}$
Typing ctx	Γ	::=	$\cdot \mid \Gamma, x : \tau$
Formula ctx	Δ	::=	$\cdot \mid \Delta, \varphi$
Exec ctx	Ξ	::=	$u_b : \mathtt{b}, u_e : \mathtt{b}, i : \mathtt{b}$

Beta reductions We define the β-reduction rules below.

$\boxed{e \to_\beta e'}$

$$\frac{}{(\lambda x.e)\ e_2 \to_\beta e[e_2/x]} \qquad \frac{}{\Lambda X.e\ \cdot \to_\beta e} \qquad \frac{e_1 \to_\beta e_1'}{e_1\ e_2 \to_\beta e_1'\ e_2}$$

$$\frac{e_1 \to_\beta e_1'}{e_1\ \cdot \to_\beta e_1'\ \cdot}$$

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

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

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

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

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

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

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

$$\frac{}{\sigma \triangleright \langle \iota; x.c_2 :: K; \mathtt{comp}(c_1) \rangle \hookrightarrow \sigma \triangleright \langle \iota; x.c_2 :: K; c_1 \rangle}\ \text{R-SEQE3}$$

$$\frac{}{\sigma \triangleright \langle \iota; K; \mathtt{letc}(c_1, x.c_2) \rangle \hookrightarrow \sigma \triangleright \langle \iota; x.c_2 :: K; c_1 \rangle}\ \text{R-SEQC}$$

$$\frac{}{\begin{array}{l}\sigma \triangleright \langle \iota; K; (\mathtt{fix} f(x).c)\ e \rangle \\ \hookrightarrow \sigma \triangleright \langle \iota; K; c[\lambda z.\mathtt{comp}(\mathtt{fix}(f(x).c)\ z)/f][e/x] \rangle\end{array}}\ \text{R-FIX}$$

$\boxed{C \to C'}$

$$\frac{\sigma \triangleright T \hookrightarrow \sigma' \triangleright T'}{\sigma \triangleright T, T_1, \ldots, T_n \to \sigma' \triangleright T', T_1, \ldots, T_n}$$

B. Well-formedness Judgments

Well-formedness judgments for contexts and types

$\boxed{\Theta \vdash \Sigma\ \mathsf{ok}}$

$$\frac{}{\Theta \vdash \cdot\ \mathsf{ok}} \qquad \frac{\Theta \vdash \Sigma\ \mathsf{ok} \qquad \Theta; \Sigma; \cdot \vdash \alpha\ \mathsf{ok}}{\Theta \vdash \Sigma, A :: \alpha\ \mathsf{ok}}$$

$\boxed{\Theta; \Sigma \vdash \Gamma\ \mathsf{ok}}$

$$\frac{\Theta \vdash \Sigma\ \mathsf{ok}}{\Theta; \Sigma \vdash \cdot\ \mathsf{ok}} \qquad \frac{\Theta; \Sigma \vdash \Gamma\ \mathsf{ok} \qquad \Theta; \Sigma; \Gamma \vdash \tau\ \mathsf{ok}}{\Theta; \Sigma \vdash \Gamma, x : \tau\ \mathsf{ok}}$$

$\boxed{\Theta; \Sigma; \Gamma \vdash \Delta\ \mathsf{ok}}$

$$\frac{\Theta; \Sigma \vdash \Gamma\ \mathsf{ok}}{\Theta; \Sigma; \Gamma \vdash \cdot\ \mathsf{ok}} \qquad \frac{\Theta; \Sigma; \Gamma \vdash \Delta\ \mathsf{ok} \qquad \Gamma \vdash \varphi\ \mathsf{ok}}{\Theta; \Sigma, \Gamma \vdash \Delta, \varphi\ \mathsf{ok}}$$

$\boxed{\Gamma \vdash \varphi\ \mathsf{ok}}$

$$\frac{}{\Gamma \vdash P\ \mathsf{ok}} \qquad \frac{\Gamma \vdash \varphi\ \mathsf{ok} \qquad \mathtt{fv}(e) \in \mathtt{dom}(\Gamma)}{\Gamma \vdash \varphi\ e\ \mathsf{ok}}$$

$$\frac{}{\Gamma \vdash \top\ \mathsf{ok}} \qquad \frac{}{\Gamma \vdash \bot\ \mathsf{ok}} \qquad \frac{\Gamma \vdash \varphi_1\ \mathsf{ok} \qquad \Gamma \vdash \varphi_2\ \mathsf{ok}}{\Gamma \vdash \varphi_1 \wedge \varphi_2\ \mathsf{ok}}$$

$$\frac{\Gamma \vdash \varphi_1\ \mathsf{ok} \qquad \Gamma \vdash \varphi_2\ \mathsf{ok}}{\Gamma \vdash \varphi_1 \vee \varphi_2\ \mathsf{ok}}$$

$$\frac{\Gamma \vdash \varphi\ \mathsf{ok}}{\Gamma \vdash \neg\varphi\ \mathsf{ok}} \qquad \frac{\tau = b \text{ or } \mathtt{any} \qquad \Gamma, x : \tau \vdash \varphi\ \mathsf{ok}}{\Gamma \vdash \forall x{:}\tau.\varphi\ \mathsf{ok}}$$

$$\frac{\tau = b \text{ or } \mathtt{any} \qquad \Gamma, x : \tau \vdash \varphi\ \mathsf{ok}}{\Gamma \vdash \exists x{:}\tau.\varphi\ \mathsf{ok}}$$

$$\frac{\mathtt{fv}(e_1) \cup \mathtt{fv}(e_2) \subseteq \mathtt{dom}(\Gamma)}{\Gamma \vdash e_1 \equiv e_2\ \mathsf{ok}}$$

$\boxed{\Theta; \Sigma; \Gamma \vdash \tau \ \mathsf{ok}}$

$$\frac{X \in \Theta \qquad \Theta; \Sigma \vdash \Gamma \ \mathsf{ok}}{\Theta; \Sigma; \Gamma \vdash X \ \mathsf{ok}}$$

$$\frac{\Theta; \Sigma; \Gamma \vdash \tau_1 \ \mathsf{ok} \qquad \Theta; \Sigma; \Gamma, x : \tau_1 \vdash \tau_2 \ \mathsf{ok}}{\Theta; \Sigma; \Gamma \vdash \Pi x{:}\tau_1.\tau_2 \ \mathsf{ok}}$$

$$\frac{\Theta; \Sigma; \Gamma \vdash \eta_c \ \mathsf{ok}}{\Theta; \Sigma; \Gamma \vdash \mathtt{comp}(\eta_c) \ \mathsf{ok}} \qquad \frac{\Theta; \Sigma \vdash \Gamma \ \mathsf{ok}}{\Theta; \Sigma; \Gamma \vdash b \ \mathsf{ok}}$$

$$\frac{\Theta, X; \Sigma; \Gamma \vdash \tau \ \mathsf{ok}}{\Theta; \Sigma; \Gamma \vdash \forall X.\tau \ \mathsf{ok}} \qquad \frac{\Theta; \Sigma \vdash \Gamma \ \mathsf{ok}}{\Theta; \Sigma; \Gamma \vdash \mathtt{any} \ \mathsf{ok}}$$

$\boxed{\Theta; \Sigma; \Gamma \vdash \alpha \ \mathsf{ok}}$

$$\frac{\Theta; \Sigma; \Gamma \vdash \eta_c \ \mathsf{ok}}{\Theta; \Sigma; \Gamma \vdash \mathtt{Act}(\eta_c) \ \mathsf{ok}}$$

$$\frac{\Theta; \Sigma; \Gamma \vdash \tau \ \mathsf{ok} \qquad \Theta; \Sigma; \Gamma, x : \tau \vdash \alpha \ \mathsf{ok}}{\Theta; \Sigma; \Gamma \vdash \Pi x{:}\tau.\alpha \ \mathsf{ok}}$$

$$\frac{\Theta, X; \Sigma; \Gamma \vdash \alpha \ \mathsf{ok}}{\Theta; \Sigma; \Gamma \vdash \forall X.\alpha \ \mathsf{ok}}$$

$\boxed{\Theta; \Sigma; \Gamma \vdash \eta_c \ \mathsf{ok}}$

$$\frac{\begin{array}{c}\Theta; \Sigma; \Gamma, u_1{:}\mathtt{b}, u_2{:}\mathtt{b}, i{:}\mathtt{b} \vdash \tau \ \mathsf{ok} \\ \Gamma, u_1{:}\mathtt{b}, u_2{:}\mathtt{b}, i{:}\mathtt{b}, x : \tau \vdash \varphi_1 \ \mathsf{ok} \qquad \Gamma, u_1{:}\mathtt{b}, u_2{:}\mathtt{b}, i{:}\mathtt{b} \vdash \varphi_2 \ \mathsf{ok}\end{array}}{\Theta; \Sigma; \Gamma \vdash u_1.u_2.i.(x{:}\tau.\varphi_1, \varphi_2) \ \mathsf{ok}}$$

$$\frac{\Theta; \Sigma; \Gamma \vdash \tau \ \mathsf{ok} \qquad \Theta; \Sigma; \Gamma, y : \tau \vdash u_1.u_2.i.(x{:}\tau_1.\varphi_1, \varphi_2) \ \mathsf{ok}}{\Theta; \Sigma; \Gamma \vdash \Pi y{:}\tau.u_1.u_2.i.(x{:}\tau_1.\varphi_1, \varphi_2) \ \mathsf{ok}}$$

C. Typing Rules

Typing for simple terms $\boxed{\Gamma \vdash_e e : \tau}$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Theta; \Upsilon \vdash \lambda x.e : \Pi x{:}\tau_1.\tau_2}$$

$$\frac{\Gamma \vdash e_1 : \Pi x{:}\tau_1.\tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 \ e_2 : \tau_2} \qquad \frac{}{\Gamma \vdash e : \mathtt{any}}$$

Confine relation

$$\frac{}{\mathit{confine}\ (b)\ (u_b.u_e.i.\varphi)}$$

$$\frac{\mathit{confine}\ (\tau_1)\ (u_b.u_e.i.\varphi) \quad \mathit{confine}\ (\tau_2)\ (u_b.u_e.i.\varphi)}{\mathit{confine}\ (\Pi\_{:}\tau_1.\tau_2)\ (u_b.u_e.i.\varphi)}$$

$$\frac{\mathit{confine}\ (\tau)\ (u_b.u_e.i.\varphi)}{\mathit{confine}\ (\mathtt{comp}(u_b.u_e.i.(x{:}\tau.\varphi, \varphi)))\ (u_b.u_e.i.\varphi)}$$

Typing rules for expressions

$\boxed{u; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash e : \tau}$

$$\frac{\Theta; \Sigma; u, \Gamma^L; \Gamma \vdash \Delta \ \mathsf{ok} \qquad x : \tau \in \Gamma}{u; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash x : \tau} \ \textsc{E-Var}$$

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

$$\frac{\Theta; \Sigma; \Gamma^L, u, \Gamma \vdash \Delta \ \mathsf{ok}}{u; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash bv : \mathtt{b}} \ \textsc{E-BaseVal}$$

$$\frac{\begin{array}{c}\Theta; \Sigma; \Gamma^L, u, \Gamma \vdash \tau_1 \ \mathsf{ok} \\ u; \Theta; \Sigma; \Gamma^L; \Gamma, x : \tau_1; \Delta \vdash_Q e : \tau_2\end{array}}{u; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_Q \lambda x.e : \Pi x{:}\tau_1.\tau_2} \ \textsc{E-Fun}$$

$$\frac{\begin{array}{c}u; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_Q e_1 : \Pi x{:}\tau_1.\tau_2 \\ u; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_Q e_2 : \tau_1\end{array}}{u; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_Q e_1 \ e_2 : \tau_2[e_2/x]} \ \textsc{E-App}$$

$$\frac{u; \Theta, X; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_Q e : \tau}{u; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_Q \Lambda X.e : \forall X.\tau} \ \textsc{E-TFun}$$

$$\frac{\begin{array}{c}u; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_Q e : \forall X.\tau_1 \\ \Theta; \Sigma; \Gamma^L, u, \Gamma \vdash \tau \ \mathsf{ok}\end{array}}{u; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_Q e \cdot : \tau_1[\tau/X]} \ \textsc{E-TApp}$$

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

$$\frac{\begin{array}{c}\varphi \text{ is trace composable} \\ u_b, u_e, i; \Theta; \Sigma; \Gamma^L, u; \Gamma; \Delta \vdash \varphi \ \mathsf{silent} \\ u_b{:}\mathtt{b}, u_e{:}\mathtt{b}, i{:}\mathtt{b} \vdash \varphi \ \mathsf{ok} \qquad \mathtt{fa}(e) = \emptyset \qquad \mathtt{fv}(e) \subseteq \Gamma \\ \mathit{confine}\ (\tau)\ (u_b.u_e.i.\varphi) \qquad \mathit{confine}\ (\Gamma)\ (u_b.u_e.i.\varphi)\end{array}}{u; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_{u_b.u_e.i.\varphi} e : \tau} \ \textsc{Confine}$$

$$\frac{u; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash e : \tau \qquad u_b{:}\mathtt{b}, u_e{:}\mathtt{b}, i{:}\mathtt{b} \vdash \varphi \ \mathsf{ok}}{u; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_{u_b.u_e.i.\varphi} e : \tau} \ \textsc{Conf-sub}$$

$$\frac{\begin{array}{l}u_1, u_2, i; \Theta; \Sigma; \Gamma^L; u_e, \Gamma; \Delta, u_1 \geq u_e \vdash_Q c : (x{:}\tau.\varphi_1, \varphi_2) \\ \Theta; \Sigma; \Gamma^L, u_e{:}\mathtt{b}, u_1{:}\mathtt{b}, u_2{:}\mathtt{b}, i{:}\mathtt{b}; \Gamma, x : \tau; \Delta \vdash \varphi_1 \Rightarrow \varphi_1' \ \mathsf{true} \\ \Theta; \Sigma; \Gamma^L, u_e{:}\mathtt{b}, u_1{:}\mathtt{b}, u_2{:}\mathtt{b}, i{:}\mathtt{b}; \Gamma; \Delta \vdash \varphi_2 \Rightarrow \varphi_2' \ \mathsf{true} \\ \Theta; \Sigma; \Gamma^L, u_e{:}\mathtt{b}; \Gamma \vdash u_1.u_2.i.(x{:}\tau.\varphi_1', \varphi_2') \ \mathsf{ok} \\ \mathtt{fv}(c) \subseteq \mathtt{dom}(\Gamma)\end{array}}{u_e; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_Q \mathtt{comp}(c) : \mathtt{comp}(u_1.u_2.i.(x{:}\tau.\varphi_1', \varphi_2'))} \ \textsc{Comp}$$

Typing rules for silent threads

$\boxed{\Xi; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash \varphi \ \mathsf{silent}}$

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

Typing rules for actions

$$\boxed{u : \mathsf{b}; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_Q a :: \alpha}$$

$$\frac{\Theta; \Sigma; \Gamma^L, u : \mathsf{b}; \Gamma \vdash \Delta\ \mathsf{ok} \qquad A :: \alpha \in \Sigma}{u : \mathsf{b}; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash A :: \alpha}$$

$$\frac{\begin{array}{c} u : \mathsf{b}; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_Q a :: \Pi x{:}\tau.\alpha \\ u : \mathsf{b}; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_Q e : \tau \end{array}}{u : \mathsf{b}; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_Q a\ e :: \alpha[e/x]}$$

$$\frac{\begin{array}{c} u : \mathsf{b}; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_Q a :: \forall X.\alpha \\ \Theta; \Sigma; \Gamma^L, u : \mathsf{b}, \Gamma \vdash \tau\ \mathsf{ok} \end{array}}{u : \mathsf{b}; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_Q a\ \cdot :: \alpha[\tau/X]}$$

Logical reasoning rules

$$\boxed{\Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash \varphi\ \mathsf{true}}$$

$$\frac{\begin{array}{c} u_1, u_2, i; \Theta; \Sigma; \Gamma^L; \cdot; \Delta \vdash c : \varphi \\ \Theta; \Sigma; \Gamma^L; \cdot; \Delta \vdash \mathsf{start}(I, c, u)\ \mathsf{true} \\ \Theta; \Sigma \vdash \Gamma^L, \Gamma\ \mathsf{ok} \end{array}}{\Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash \forall u'{:}\mathsf{b}.(u' > u) \Rightarrow \varphi[u, u', I/u_1, u_2, i]\ \mathsf{true}}\ \textsc{Honest}$$

$$\frac{\begin{array}{c} \Theta; \Sigma; \Gamma^L; \Gamma; \Delta_1 \vdash \varphi\ \mathsf{true} \\ \Theta; \Sigma; \Gamma^L; \Gamma; \Delta_1, \varphi, \Delta_2 \vdash \varphi'\ \mathsf{true} \end{array}}{\Theta; \Sigma; \Gamma^L; \Gamma; \Delta_1, \Delta_2 \vdash \varphi'\ \mathsf{true}}\ \textsc{Cut}$$

$$\frac{\Theta; \Sigma; \Gamma^L, \Gamma \vdash \Delta\ \mathsf{ok} \qquad \varphi \in \Delta}{\Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash \varphi\ \mathsf{true}}\ \textsc{Init}$$

$$\frac{\Theta; \Sigma; \Gamma^L; \Gamma; \Delta_1, \varphi, \Delta_2 \vdash \cdot}{\Theta; \Sigma; \Gamma^L; \Gamma; \Delta_1, \Delta_2 \vdash \neg\varphi\ \mathsf{true}}\ \neg\mathrm{I}$$

$$\frac{\Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash \neg\varphi\ \mathsf{true}}{\Theta; \Sigma; \Gamma^L; \Gamma; \Delta, \varphi \vdash \cdot}\ \neg\mathrm{E}$$

$$\frac{\Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash \varphi_1\ \mathsf{true} \qquad \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash \varphi_2\ \mathsf{true}}{\Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash \varphi_1 \wedge \varphi_2\ \mathsf{true}}\ \wedge\mathrm{I}$$

$$\frac{i \in [1,2], \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash \varphi_1 \wedge \varphi_2\ \mathsf{true}}{\Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash \varphi_i\ \mathsf{true}}\ \wedge\mathrm{E}$$

$$\frac{i \in [1,2], \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash \varphi_i\ \mathsf{true}}{\Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash \varphi_1 \vee \varphi_2\ \mathsf{true}}\ \vee\mathrm{I}$$

$$\frac{\begin{array}{c} \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash \varphi_1 \vee \varphi_2\ \mathsf{true} \\ \Theta; \Sigma; \Gamma^L; \Gamma; \Delta, \varphi_1, \Gamma' \vdash \varphi\ \mathsf{true} \\ \Theta; \Sigma; \Gamma^L; \Gamma; \Delta, \varphi_2, \Gamma' \vdash \varphi\ \mathsf{true} \end{array}}{\Theta; \Sigma; \Gamma^L; \Gamma; \Delta, \Gamma' \vdash \varphi\ \mathsf{true}}\ \vee\mathrm{E}$$

$$\frac{\Theta; \Sigma; \Gamma^L, x : \tau; \Gamma; \Delta \vdash \varphi\ \mathsf{true}}{\Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash \forall x{:}\tau.\varphi\ \mathsf{true}}\ \forall\mathrm{I}$$

$$\frac{\Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash \forall x{:}\tau.\varphi\ \mathsf{true} \qquad \Gamma^L \vdash t : \tau}{\Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash \varphi[t/x]\ \mathsf{true}}\ \forall\mathrm{E}$$

$$\frac{\Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash \varphi[t/x]\ \mathsf{true} \qquad \Gamma^L \vdash t : \tau}{\Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash \exists x{:}\tau.\varphi\ \mathsf{true}}\ \exists\mathrm{I}$$

$$\frac{\begin{array}{c} \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash \exists x{:}\tau.\varphi\ \mathsf{true} \\ \Theta; \Sigma; \Gamma^L, a{:}\tau; \Gamma; \Delta, \varphi[a/x] \vdash \varphi'\ \mathsf{true} \qquad a \notin \mathtt{fv}(\varphi') \end{array}}{\Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash \varphi'\ \mathsf{true}}\ \exists\mathrm{E}$$

Typing rules for computations We summarize the typing rules for computations in Figures 8 and 9.

D. Semantics

Semantics for invariant properties Next we define a logical relation indexed only by an invariant property $u_b.u_e.i.\varphi$.

$$\begin{array}{l} \mathcal{RV}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u} = \{(k, \mathtt{nf}) \mid \mathtt{nf} \neq \lambda x.e, \Lambda X.e, \mathtt{comp}(c)\} \\ \cup\{(k, \mathtt{comp}(c)) \mid (k, c) \in \mathcal{RC}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}\} \\ \cup\{(k, \lambda x.e') \mid \forall j, u', j < k, u' \geq u \\ \qquad\qquad (j, e') \in \mathcal{RE}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u'} \\ \qquad\qquad \Longrightarrow (j, e[e'/x]) \in \mathcal{RE}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u'}\} \\ \cup \{(k, \Lambda x.e) \mid \forall j, j < k \Longrightarrow (j, e) \in \mathcal{RE}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}\} \end{array}$$

$$\begin{array}{l} \mathcal{RE}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u} = \\ \{(k, e) \mid \forall 0 \leq m \leq k, e \rightarrow^m e' \not\rightarrow \\ \qquad \Longrightarrow (n - m, e') \in \mathcal{RV}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}\} \end{array}$$

$$\begin{array}{l} \mathcal{RC}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u} = \{(k, c) \mid \\ \quad \forall u_B, u_E, \iota, u \leq u_B \leq u_E, \text{let } \gamma = [u_B, u_E, \iota/u_1, u_2, i], \\ \quad j_b \text{ is the length of the trace from time } u_B \text{ to the end of } \mathcal{T}, \\ \quad j_e \text{ is the length of the trace from time } u_E \text{ to the end of } \mathcal{T} \\ \quad k \geq j_b \geq j_e, \\ \quad \text{the configuration at time } u_B \text{ is } \xrightarrow{u_B} \sigma_b \triangleright \cdots, \langle \iota; x.c' :: K; c\rangle \cdots \\ \quad \text{between } u_B \text{ and } u_E \text{ (inclusive), the stack of thread } i \text{ always} \\ \quad \text{contains prefix } x.c'{::}K \\ \quad \Longrightarrow \mathcal{T} \vDash_\theta \varphi\} \cap \\ \{(k, c) \mid \forall u_B, u_E, \iota, u \leq u_B \leq u_E, \\ \quad \text{let } \gamma = [u_B, u_E, \iota/u_1, u_2, i], \\ \quad j_b \text{ is the length of the trace from time } u_B \text{ to the end of } \mathcal{T} \\ \quad j_e \text{ is the length of the trace from time } u_E \text{ to the end of } \mathcal{T} \\ \quad k \geq j_b > j_e, \\ \quad \text{the configuration at time } u_1 \text{ is } \xrightarrow{u_B} \sigma_b \triangleright \cdots, \langle \iota; x.c' :: K; c\rangle \cdots \\ \quad \text{the configuration at time } u_E \text{ is } \xrightarrow{u_E} \sigma_e \triangleright \cdots, \langle \iota; K; c'[e'/x]\rangle \cdots \\ \quad \text{between } u_B \text{ and } u_E, \text{ the stack of thread } i \text{ always contains } x.c'{::}K \\ \quad \Longrightarrow (j_e, e') \in \mathcal{RE}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u_E} \text{ and } \mathcal{T} \vDash_\theta \varphi[e'/x]\} \end{array}$$

Fixpoint $\boxed{u : \mathtt{b}; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_Q c : \eta}$

$$\frac{\begin{array}{l}\Gamma_1 = y : \tau, f : \Pi y{:}\tau.\mathtt{comp}(u_1.u_3.i.(x{:}\tau_1.\varphi, \varphi')) \\ u_1 : \mathtt{b}, u_2 : \mathtt{b}, i : \mathtt{b}; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta, u \le u_1 \le u_2 \vdash \varphi_0 \ \mathsf{silent} \\ u_2, u_3, i; \Theta; \Sigma; \Gamma^L, u_1 : \mathtt{b}, u : \mathtt{b}; \Gamma, \Gamma_1; \Delta, u_2 < u_3, \varphi_0 \vdash_Q c : x{:}\tau_1.\varphi_1 \\ u_2, u_3, i; \Theta; \Sigma; \Gamma^L; u_1 : \mathtt{b}, u : \mathtt{b}; \Gamma, \Gamma_1; \Delta, u_2 \le u_3, \varphi_0 \vdash_Q c : \varphi_2 \\ \Theta; \Sigma; \Gamma^L, u_1 : \mathtt{b}, u : \mathtt{b}, u_2 : \mathtt{b}, u_3 : \mathtt{b}, i : \mathtt{b}; \Gamma, \Gamma_1, x : \tau_1; \Delta \vdash (\varphi_0 \wedge \varphi_1) \Rightarrow \varphi \ \mathsf{true} \\ \Theta; \Sigma; \Gamma^L, u_1 : \mathtt{b}, u_2 : \mathtt{b}, u_3 : \mathtt{b}, i : \mathtt{b}, u : \mathtt{b}; \Gamma, \Gamma_1; \Delta \vdash (\varphi_0 \wedge \varphi_2 \Rightarrow \varphi') \ \mathsf{true} \\ \Theta; \Sigma; \Gamma^L, u_1 : \mathtt{b}, u_3 : \mathtt{b}, i : \mathtt{b}, u : \mathtt{b}; \Gamma, y : \tau; \Delta \vdash \varphi_0[u_3/u_2] \Rightarrow \varphi' \ \mathsf{true} \\ \Theta; \Sigma; \Gamma^L, u : \mathtt{b}; \Gamma \vdash \Pi y{:}\tau.u_1.u_3.i.(x{:}\tau_1.\varphi, \varphi') \ \mathsf{ok} \qquad \mathtt{fv}(\mathtt{fix}(f(y).c)) \in \mathtt{dom}(\Gamma)\end{array}}{u; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_Q \mathtt{fix}(f(y).c) : \Pi y{:}\tau.u_1.u_3.i.(x{:}\tau_1.\varphi, \varphi')} \textsc{Fix}$$

Partial correctness typing $\boxed{\Xi; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_Q c : \eta}$

$$\frac{\begin{array}{l}u_1 : \mathtt{b}; \Theta; \Sigma; \Gamma^L, u_2 : \mathtt{b}, i : \mathtt{b}; \Gamma; \Delta \vdash_Q c : \Pi y{:}\tau.u_b.u_e.j.(x{:}\tau'.\varphi, \varphi') \qquad u_1 : \mathtt{b}; \Theta; \Sigma; \Gamma^L, u_2 : \mathtt{b}, i : \mathtt{b}; \Gamma; \Delta \vdash_Q e : \tau \\ \mathtt{fv}(c\ e) \subseteq \mathtt{dom}(\Gamma) \qquad \text{let } \gamma = [u_1, u_2, i/u_b, u_e, j] \qquad \Theta; \Sigma; \Gamma^L, \Gamma \vdash u_1.u_2.i.((x{:}\tau'.\varphi)\gamma[e/y], \varphi'\gamma[e/y]) \ \mathsf{ok}\end{array}}{u_1 : \mathtt{b}, u_2 : \mathtt{b}, i : \mathtt{b}; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_Q c\ e : ((x{:}\tau'\gamma.\varphi\gamma)[e/y], \varphi'\gamma[e/y])} \textsc{App}$$

$$\frac{\begin{array}{l}u_1 : \mathtt{b}; \Theta; \Sigma; \Gamma^L, u_2 : \mathtt{b}, i : \mathtt{b}; \Gamma; \Delta \vdash_Q a :: \mathtt{Act}(u_b.u_e.j.(x{:}\tau.\varphi_1, \varphi_2)) \qquad u_1 : \mathtt{b}, u_2 : \mathtt{b}, i : \mathtt{b}; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash \varphi \ \mathsf{silent} \\ \mathtt{fv}(a) \in \mathtt{dom}(\Gamma) \qquad \Theta; \Sigma; \Gamma^L; \Gamma \vdash u_1.u_2.i.(x{:}\tau.\varphi_1[u_1, u_2, i/u_b, u_e, j], \varphi_2[u_1, u_2, i/u_b, u_e, j] \wedge \varphi) \ \mathsf{ok}\end{array}}{u_1 : \mathtt{b}, u_2 : \mathtt{b}, i : \mathtt{b}; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_Q \mathtt{act}(a) : (x{:}\tau.\varphi_1[u_1, u_2, i/u_b, u_e, j], \varphi_2[u_1, u_2, i/u_b, u_e, j] \wedge \varphi)} \textsc{Act}$$

$$\frac{\begin{array}{l}u_0 : \mathtt{b}, u_1 : \mathtt{b}, i : \mathtt{b}; \Theta; \Sigma; \Gamma^L, u_3 : \mathtt{b}; \Gamma; \Delta, u_0 \le u_1 \vdash \varphi_0 \ \mathsf{silent} \\ u_1 : \mathtt{b}; \Theta; \Sigma; \Gamma^L, u_0 : \mathtt{b}, u_3 : \mathtt{b}, i : \mathtt{b}; \Gamma; \Delta, \varphi_0 \vdash_Q e_1 : \mathtt{comp}(u_b, u_e, j.(x{:}\tau.\varphi_1, \varphi_1')) \\ \text{let } \gamma = [u_1, u_2, i/u_b, u_e, j] \\ u_2 : \mathtt{b}, u_3 : \mathtt{b}, i : \mathtt{b}; \Theta; \Sigma; \Gamma^L, u_0 : \mathtt{b}, u_1 : \mathtt{b}; \Gamma, x : \tau\gamma; \Delta, u_2 < u_3, \varphi_0, \varphi_1\gamma \vdash_Q c_2 : y{:}\tau'.\varphi_2 \\ \Theta; \Sigma; \Gamma^L, u_0 : \mathtt{b}, u_3 : \mathtt{b}, i : \mathtt{b}; \Gamma, u_1{:}\mathtt{b}, u_2{:}\mathtt{b}, x{:}\tau\gamma, y : \tau'; \Delta \vdash (\varphi_0 \wedge \varphi_1\gamma \wedge \varphi_2) \Rightarrow \varphi \ \mathsf{true} \\ \mathtt{fv}(\mathtt{lete}(e_1, x.c_2)) \subseteq \mathtt{dom}(\Gamma) \qquad \Theta; \Sigma; \Gamma^L, u_0 : \mathtt{b}, u_3 : \mathtt{b}, i : \mathtt{b}; \Gamma, y : \tau' \vdash \varphi \ \mathsf{ok}\end{array}}{u_0 : \mathtt{b}, u_3 : \mathtt{b}, i : \mathtt{b}; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_Q \mathtt{lete}(e_1, x.c_2) : y{:}\tau'.\varphi} \textsc{SeqE}$$

$$\frac{\begin{array}{l}u_0 : \mathtt{b}, u_1 : \mathtt{b}, i : \mathtt{b}; \Theta; \Sigma; \Gamma^L; u_3 : \mathtt{b}, \Gamma; \Delta, u_0 \le u_1 \vdash \varphi_0 \ \mathsf{silent} \\ u_1 : \mathtt{b}, u_2 : \mathtt{b}, i : \mathtt{b}; \Theta; \Sigma; \Gamma^L, u_0 : \mathtt{b}, u_3 : \mathtt{b}; \Gamma; \Delta, u_1 < u_2, \varphi_0 \vdash_Q c_1 : x{:}\tau.\varphi_1 \\ u_2 : \mathtt{b}, u_3 : \mathtt{b}, i : \mathtt{b}; \Theta; \Sigma; \Gamma^L, u_0 : \mathtt{b}, u_1 : \mathtt{b}; \Gamma, x : \tau; \Delta, u_2 < u_3, \varphi_0, \varphi_1 \vdash_Q c_2 : y{:}\tau'.\varphi_2 \\ \Theta; \Sigma; \Gamma^L, u_1{:}\mathtt{b}, u_2{:}\mathtt{b}, u_0 : \mathtt{b}, u_3 : \mathtt{b}, i : \mathtt{b}; \Gamma, x{:}\tau, y : \tau'; \Delta \vdash (\varphi_0 \wedge \varphi_1 \wedge \varphi_2) \Rightarrow \varphi \ \mathsf{true} \\ \Theta; \Sigma; \Gamma^L, u_0 : \mathtt{b}, u_3 : \mathtt{b}, i : \mathtt{b}; \Gamma, y : \tau' \vdash \varphi \ \mathsf{ok} \qquad \mathtt{fv}(\mathtt{letc}(c_1, x.c_2)) \subseteq \mathtt{dom}(\Gamma)\end{array}}{u_0 : \mathtt{b}, u_3 : \mathtt{b}, i : \mathtt{b}; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_Q \mathtt{letc}(c_1, x.c_2) : y{:}\tau'.\varphi} \textsc{SeqC}$$

$$\frac{u_2 : \mathtt{b}; \Theta; \Sigma; \Gamma^L, u_1 : \mathtt{b}, i : \mathtt{b}; \Gamma; \Delta \vdash_Q e : \tau \qquad u_1 : \mathtt{b}, u_2 : \mathtt{b}, i : \mathtt{b}; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash \varphi \ \mathsf{silent} \qquad \mathtt{fv}(e) \subseteq \mathtt{dom}(\Gamma)}{u_1 : \mathtt{b}, u_2 : \mathtt{b}, i : \mathtt{b}; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_Q \mathtt{ret}(e) : x{:}\tau.((x \equiv e) \wedge \varphi)} \textsc{Ret}$$

$$\frac{\begin{array}{l}u_0 : \mathtt{b}, u_1 : \mathtt{b}, i : \mathtt{b}; \Theta; \Sigma; \Gamma^L, u_2 : \mathtt{b}; \Gamma; \Delta, u_0 \le u_1 \vdash \varphi_0 \ \mathsf{silent} \\ u_0 : \mathtt{b}; \Theta; \Sigma; \Gamma^L, u_2 : \mathtt{b}, i : \mathtt{b}; \Gamma; \Delta \vdash_Q e : \mathtt{b} \qquad u_1 : \mathtt{b}, u_2 : \mathtt{b}, i : \mathtt{b}; \Theta; \Sigma; \Gamma^L, u_0 : \mathtt{b}; \Gamma; \Delta, u_1 < u_2, \varphi_0, (\mathsf{eval}\, e\, \mathtt{tt}) \vdash_Q c_1 : x{:}\tau.\varphi_1 \\ u_1 : \mathtt{b}, u_2 : \mathtt{b}, i : \mathtt{b}; \Theta; \Sigma; \Gamma^L, u_0 : \mathtt{b}; \Gamma; \Delta, u_1 < u_2, \varphi_0, (\mathsf{eval}\, e\, \mathtt{ff}) \vdash_Q c_2 : x{:}\tau.\varphi_2 \\ \Theta; \Sigma; \Gamma^L, u_0 : \mathtt{b}, u_2 : \mathtt{b}, i : \mathtt{b}; \Gamma, u_1 : \mathtt{b}, x : \tau; \Delta \vdash (\varphi_0 \wedge \varphi_i) \Rightarrow \varphi \text{ where } i \in [1, 2] \\ \Theta; \Sigma; \Gamma^L, u_0 : \mathtt{b}, u_2 : \mathtt{b}, i : \mathtt{b}; \Gamma, x : \tau \vdash \varphi \ \mathsf{ok} \qquad \mathtt{fv}(e) \cup \mathtt{fv}(c_1) \cup \mathtt{fv}(c_2) \subseteq \mathtt{dom}(\Gamma)\end{array}}{u_0 : \mathtt{b}, u_2 : \mathtt{b}, i : \mathtt{b}; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_Q \mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 : x{:}\tau.\varphi} \textsc{If}$$

Figure 8. Computation typing rules (1)

Invariant typing $\boxed{\Xi; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_Q c : \eta}$

$$\frac{\begin{array}{l}\Theta; \Sigma; \Gamma^L, u_0 : \mathtt{b}, u_3 : \mathtt{b}, i : \mathtt{b}; \Gamma; \Delta \vdash \varphi \ \mathsf{ok} \\ u_0 : \mathtt{b}, u_1 : \mathtt{b}, i : \mathtt{b}; \Theta; \Sigma; \Gamma^L, u_3 : \mathtt{b}; \Gamma; \Delta, u_0 \le u_1 \vdash \varphi_0 \ \mathsf{silent} \qquad u_0 : \mathtt{b}, u_3 : \mathtt{b}, i : \mathtt{b}; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta, u_0 \le u_3 \vdash \varphi_0' \ \mathsf{silent} \\ u_1 : \mathtt{b}; \Theta; \Sigma; \Gamma^L, u_0 : \mathtt{b}, u_3 : \mathtt{b}, i : \mathtt{b}; \Gamma; \Delta, \varphi_0 \vdash_Q e_1 : \mathtt{comp}(u_b, u_e, j.(x{:}\tau.\varphi_1, \varphi_1')) \\ u_2 : \mathtt{b}, u_3 : \mathtt{b}, i : \mathtt{b}; \Theta; \Sigma; \Gamma^L, u_0 : \mathtt{b}; \Gamma; \Delta, u_1 : \mathtt{b}, x : \tau; u_1 < u_2 \le u_3, \varphi_0, \varphi_1[u_1, u_2, i/u_b, u_e, j] \vdash_Q c_2 : \varphi_2 \\ \Theta; \Sigma; \Gamma^L, u_0 : \mathtt{b}, u_3 : \mathtt{b}, i : \mathtt{b}; \Gamma; \Delta \vdash \varphi_0' \Rightarrow \varphi \ \mathsf{true} \\ \Theta; \Sigma; \Gamma^L, u_0 : \mathtt{b}, u_3 : \mathtt{b}, i : \mathtt{b}; \Gamma, u_1{:}\mathtt{b}; \Delta \vdash \varphi_0 \wedge \varphi_1'[u_1, u_3, i/u_b, u_e, j] \Rightarrow \varphi \ \mathsf{true} \\ \Theta; \Sigma; \Gamma^L, u_0 : \mathtt{b}, u_3 : \mathtt{b}, i : \mathtt{b}; \Gamma, u_1{:}\mathtt{b}, u_2{:}\mathtt{b}, x{:}\tau; \Delta \vdash (\varphi_0 \wedge \varphi_1[u_1, u_2, i/u_b, u_e, j] \wedge \varphi_2) \Rightarrow \varphi \ \mathsf{true} \\ \mathtt{fv}(\mathtt{lete}(e_1, x.c_2)) \subseteq \mathtt{dom}(\Gamma)\end{array}}{u_0 : \mathtt{b}, u_3 : \mathtt{b}, i : \mathtt{b}; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_Q \mathtt{lete}(e_1, x.c_2) : \varphi} \ \textsc{SeqEI}$$

$$\frac{\begin{array}{l}\Theta; \Sigma; \Gamma^L, u_0 : \mathtt{b}, u_3 : \mathtt{b}, i : \mathtt{b}; \Gamma; \Delta \vdash \varphi \ \mathsf{ok} \\ u_0 : \mathtt{b}, u_1 : \mathtt{b}, i : \mathtt{b}; \Theta; \Sigma; \Gamma^L, u_3 : \mathtt{b}; \Gamma; \Delta, u_0 \le u_1 \vdash \varphi_0 \ \mathsf{silent} \qquad u_0 : \mathtt{b}, u_3 : \mathtt{b}, i : \mathtt{b}; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta, u_0 \le u_3 \vdash \varphi_0' \ \mathsf{silent} \\ u_1 : \mathtt{b}, u_2 : \mathtt{b}, i : \mathtt{b}; \Theta; \Sigma; \Gamma^L, u_0 : \mathtt{b}, u_3 : \mathtt{b}; \Gamma; \Delta, u_1 < u_2, \varphi_0 \vdash_Q c_1 : x{:}\tau.\varphi_1 \\ u_1 : \mathtt{b}, u_3 : \mathtt{b}, i : \mathtt{b}; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta, u_0 : \mathtt{b}, u_1 \le u_3, \varphi_0 \vdash_Q c_1 : \varphi_1' \\ u_2 : \mathtt{b}, u_3 : \mathtt{b}, i : \mathtt{b}; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta, u_0 : \mathtt{b}, u_1 : \mathtt{b}, x : \tau, u_2 \le u_3, \varphi_0, \varphi_1 \vdash_Q c_2 : \varphi_2 \\ \Theta; \Sigma; \Gamma^L, u_0 : \mathtt{b}, u_3 : \mathtt{b}, i : \mathtt{b}; \Gamma; \Delta \vdash \varphi_0' \Rightarrow \varphi \ \mathsf{true} \\ \Theta; \Sigma; \Gamma^L, u_0 : \mathtt{b}, u_3 : \mathtt{b}, i : \mathtt{b}; \Gamma, u_1{:}\mathtt{b}; \Delta \vdash (\varphi_0 \wedge \varphi_1') \Rightarrow \varphi \ \mathsf{true} \\ \Theta; \Sigma; \Gamma^L, u_0 : \mathtt{b}, u_3 : \mathtt{b}, i : \mathtt{b}; \Gamma, u_1{:}\mathtt{b}, u_2{:}\mathtt{b}, x{:}\tau; \Delta \vdash (\varphi_0 \wedge \varphi_1 \wedge \varphi_2) \Rightarrow \varphi \ \mathsf{true} \qquad \mathtt{fv}(\mathtt{letc}(c_1, x.c_2)) \subseteq \mathtt{dom}(\Gamma)\end{array}}{u_0 : \mathtt{b}, u_3 : \mathtt{b}, i : \mathtt{b}; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_Q \mathtt{letc}(c_1, x.c_2) : \varphi} \ \textsc{SeqCI}$$

$$\frac{\mathtt{fv}(e) \subseteq \mathtt{dom}(\Gamma) \qquad u_1 : \mathtt{b}, u_2 : \mathtt{b}, i : \mathtt{b}; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash \varphi \ \mathsf{silent}}{u_1 : \mathtt{b}, u_2 : \mathtt{b}, i : \mathtt{b}; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_Q \mathtt{ret}(e) : \varphi} \ \textsc{RetI}$$

$$\frac{\begin{array}{l}\Theta; \Sigma; \Gamma^L, u_0 : \mathtt{b}, u_2 : \mathtt{b}, i : \mathtt{b}; \Gamma; \Delta \vdash_Q e : \mathtt{b} \\ u_0 : \mathtt{b}, u_2 : \mathtt{b}, i : \mathtt{b}; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta, u_1 \le u_2 \vdash \varphi_0 \ \mathsf{silent} \qquad u_0 : \mathtt{b}, u_1 : \mathtt{b}, i : \mathtt{b}; \Theta; \Sigma; \Gamma^L, u_2 : \mathtt{b}; \Gamma; \Delta, u_0 \le u_1 \vdash \varphi_0' \ \mathsf{silent} \\ u_1 : \mathtt{b}, u_2 : \mathtt{b}, i : \mathtt{b}; \Theta; \Sigma; \Gamma^L, u_0 : \mathtt{b}; \Gamma; \Delta, u_1 \le u_2, \varphi_0', (\mathsf{eval}\, e\, \mathtt{tt}) \vdash_Q c_1 : \varphi_1 \\ u_1 : \mathtt{b}, u_2 : \mathtt{b}, i : \mathtt{b}; \Theta; \Sigma; \Gamma^L, u_0 : \mathtt{b}; \Gamma; \Delta, u_1 \le u_2, \varphi_0', (\mathsf{eval}\, e\, \mathtt{ff}) \vdash_Q c_2 : \varphi_2 \qquad \Theta; \Sigma; \Gamma^L, u_0 : \mathtt{b}, u_2 : \mathtt{b}, i : \mathtt{b}; \Gamma; \Delta \vdash \varphi_0 \Rightarrow \varphi \\ \Theta; \Sigma; \Gamma^L, u_0 : \mathtt{b}, u_2 : \mathtt{b}, i : \mathtt{b}; \Gamma, u_1 : \mathtt{b}; \Delta \vdash (\varphi_0' \wedge \varphi_1) \Rightarrow \varphi \qquad \Theta; \Sigma; \Gamma^L, u_0 : \mathtt{b}, u_2 : \mathtt{b}, i : \mathtt{b}; \Gamma, u_1 : \mathtt{b}; \Delta \vdash (\varphi_0' \wedge \varphi_2) \Rightarrow \varphi \\ \Theta; \Sigma; \Gamma^L, u_0 : \mathtt{b}, u_2 : \mathtt{b}, i : \mathtt{b}; \Gamma; \Delta \vdash \varphi \ \mathsf{ok} \qquad \mathtt{fv}(e) \cup \mathtt{fv}(c_1) \cup \mathtt{fv}(c_2) \subseteq \mathtt{dom}(\Gamma)\end{array}}{u_0 : \mathtt{b}, u_2 : \mathtt{b}, i : \mathtt{b}; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_Q \mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 : \varphi} \ \textsc{IfI}$$

$$\frac{u_1 : \mathtt{b}, u_2 : \mathtt{b}, i : \mathtt{b}; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta, \varphi_1 \vdash_Q c : \varphi_2}{u_1 : \mathtt{b}, u_2 : \mathtt{b}, i : \mathtt{b}; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_Q c : \varphi_1 \Rightarrow \varphi_2} \ \textsc{ImpI}$$

Misc

$$\frac{k \in [1, 2] \qquad u_1, u_2, i; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_Q c : (\eta_1, \eta_2)}{u_1, u_2, i; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_Q c : \eta_k} \ \textsc{Pair}$$

$$\frac{u_1, u_2, i; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_Q c : x{:}\tau.\varphi_1 \qquad u_1, u_2, i; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_Q c : \varphi_2}{u_1, u_2, i; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_Q c : (x{:}\tau.\varphi_1, \varphi_2)} \ \textsc{Proj}$$

$$\frac{\Theta; \Sigma; \Gamma^L, \Xi; \Gamma; \Delta_1 \vdash \varphi \ \mathsf{true} \qquad \Xi; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta_1, \varphi, \Delta_2 \vdash_Q c : \eta}{\Xi; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta_1, \Delta_2 \vdash_Q c : \eta} \ \textsc{CutC}$$

Figure 9. Computation typing (2)

$$\frac{\begin{array}{l}u_0:\mathtt{b}, u_2:\mathtt{b}, i:\mathtt{b}; \Theta; \Sigma; \Gamma^L, u_3:\mathtt{b}; \cdot; \Delta, u_0 \le u_1 \le u_2 \vdash \varphi_0 \ \mathsf{silent}\\ u_1:\mathtt{b}; \Theta; \Sigma; \Gamma^L, u_0:\mathtt{b}, u_2:\mathtt{b}, u_3:\mathtt{b}, i:\mathtt{b}; \cdot; \varphi_0 \vdash_{Q1} e_1 : \mathtt{comp}(u_b, u_e, j.(x{:}\tau.\varphi_1, \varphi_1'))\\ \text{let } \gamma = [u_1, u_2, i/u_b, u_e, j]\\ u_2, u_3, i; \Theta; \Sigma; \Gamma^L, u_0:\mathtt{b}, u_1:\mathtt{b}; \cdot; \Delta, u_2 < u_3, \varphi_0, \varphi_1\gamma \vdash_{Q2} c_2 : y{:}\tau'.\varphi_2\\ \Theta; \Sigma; \Gamma^L, u_0:\mathtt{b}, u_3:\mathtt{b}, i:\mathtt{b}; \Gamma, u_1{:}\mathtt{b}, u_2{:}\mathtt{b}, y:\tau'; \Delta \vdash (\varphi_0 \wedge \varphi_1\gamma \wedge \varphi_2) \Rightarrow \varphi \ \mathsf{true}\\ \Theta; \Sigma; \Gamma^L, u_0:\mathtt{b}, u_3:\mathtt{b}, i:\mathtt{b}, \Gamma, y:\tau' \vdash \varphi \ \mathsf{ok}\end{array}}{u_0:\mathtt{b}, u_3:\mathtt{b}, i:\mathtt{b}; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_{Q2} (e_1; c_2) : y{:}\tau'.\varphi}\ \textsc{SeqEComp}$$

$$\frac{\begin{array}{l}u_0:\mathtt{b}, u_1:\mathtt{b}, i:\mathtt{b}; \Theta; \Sigma; \Gamma^L, u_3:\mathtt{b}; \cdot; \Delta, u_0 \le u_1 \vdash \varphi_0 \ \mathsf{silent}\\ u_1:\mathtt{b}, u_2:\mathtt{b}, i:\mathtt{b}; \Theta; \Sigma; \Gamma^L, u_0:\mathtt{b}, u_3:\mathtt{b}; \cdot; \varphi_0 \vdash_Q c_1 : x{:}\tau.\varphi_1\\ u_2:\mathtt{b}, u_3:\mathtt{b}, i:\mathtt{b}; \Theta; \Sigma; \Gamma^L; u_0:\mathtt{b}, u_1:\mathtt{b}, \cdot; \Delta, u_2 < u_3, \varphi_0, \varphi_1 \vdash_{Q2} c_2 : y{:}\tau'.\varphi_2\\ \Theta; \Sigma; \Gamma^L; u_0:\mathtt{b}, u_3:\mathtt{b}, i:\mathtt{b}; \cdot, u_1{:}\mathtt{b}, u_2{:}\mathtt{b}, y:\tau'; \Delta \vdash (\varphi_0 \wedge \varphi_1 \wedge \varphi_2) \Rightarrow \varphi \ \mathsf{true}\\ \Theta; \Sigma; \Gamma^L; u_0:\mathtt{b}, u_3:\mathtt{b}, i:\mathtt{b}, \Gamma, y:\tau' \vdash \varphi \ \mathsf{ok}\end{array}}{u_0:\mathtt{b}, u_3:\mathtt{b}, i:\mathtt{b}; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_{Q2} (c_1; c_2) : y{:}\tau'.\varphi}\ \textsc{SeqCComp}$$

$$\frac{\begin{array}{l}\Theta; \Sigma; \Gamma^L, u_0:\mathtt{b}, u_3:\mathtt{b}, i:\mathtt{b}; \Gamma; \Delta \vdash \varphi \ \mathsf{ok} \qquad u_0:\mathtt{b}, u_2:\mathtt{b}, i:\mathtt{b}; \Theta; \Sigma; \Gamma^L, u_3:\mathtt{b}; \cdot; \Delta, u_0 \le u_1 \le u_2 \vdash \varphi_0 \ \mathsf{silent}\\ u_0:\mathtt{b}, u_3:\mathtt{b}, i:\mathtt{b}; \Theta; \Sigma; \Gamma^L; \cdot; \Delta, u_0 \le u_3 \vdash \varphi_0' \ \mathsf{silent}\\ u_1:\mathtt{b}; \Theta; \Sigma; \Gamma^L, u_0:\mathtt{b}, u_2:\mathtt{b}, u_3:\mathtt{b}, i:\mathtt{b}; \cdot; \varphi_0 \vdash_Q e_1 : \mathtt{comp}(u_b, u_e, j.(x{:}\tau.\varphi_1, \varphi_1'))\\ u_2:\mathtt{b}, u_3:\mathtt{b}, i:\mathtt{b}; \Theta; \Sigma; \Gamma^L, u_0:\mathtt{b}; \cdot; \Delta, u_1:\mathtt{b}; u_1 < u_2 \le u_3, \varphi_0, \varphi_1[u_1, u_2, i/u_b, u_e, j] \vdash_Q c_2 : \varphi_2\\ \Theta; \Sigma; \Gamma^L, u_0:\mathtt{b}, u_3:\mathtt{b}, i:\mathtt{b}; \Gamma; \Delta \vdash \varphi_0' \Rightarrow \varphi \ \mathsf{true}\\ \Theta; \Sigma; \Gamma^L, u_0:\mathtt{b}, u_3:\mathtt{b}, i:\mathtt{b}; \Gamma, u_1{:}\mathtt{b}; \Delta \vdash \varphi_0 \wedge \varphi_1'[u_1, u_3, i/u_b, u_e, j] \Rightarrow \varphi \ \mathsf{true}\\ \Theta; \Sigma; \Gamma^L, u_0:\mathtt{b}, u_3:\mathtt{b}, i:\mathtt{b}; \Gamma, u_1{:}\mathtt{b}, u_2{:}\mathtt{b}; \Delta \vdash (\varphi_0 \wedge \varphi_1[u_1, u_2, i/u_b, u_e, j] \wedge \varphi_2) \Rightarrow \varphi \ \mathsf{true}\end{array}}{u_0:\mathtt{b}, u_3:\mathtt{b}, i:\mathtt{b}; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_Q (e_1; c_2) : \varphi}\ \textsc{SeqEIComp}$$

$$\frac{\begin{array}{l}\Theta; \Sigma; \Gamma^L, u_0:\mathtt{b}, u_3:\mathtt{b}, i:\mathtt{b}; \Gamma; \Delta \vdash \varphi \ \mathsf{ok} \qquad u_0:\mathtt{b}, u_1:\mathtt{b}, i:\mathtt{b}; \Theta; \Sigma; \Gamma^L, u_3:\mathtt{b}; \cdot; \Delta, u_0 \le u_1 \vdash \varphi_0 \ \mathsf{silent}\\ u_0:\mathtt{b}, u_3:\mathtt{b}, i:\mathtt{b}; \Theta; \Sigma; \Gamma^L; \cdot; \Delta, u_0 \le u_3 \vdash \varphi_0' \ \mathsf{silent}\\ u_1:\mathtt{b}, u_2:\mathtt{b}, i:\mathtt{b}; \Theta; \Sigma; \Gamma^L, u_0:\mathtt{b}, u_3:\mathtt{b}; \cdot; \varphi_0 \vdash_Q c_1 : x{:}\tau.\varphi_1\\ u_1:\mathtt{b}, u_3:\mathtt{b}, i:\mathtt{b}; \Theta; \Sigma; \Gamma^L; \cdot; \Delta, u_0:\mathtt{b}, u_1 \le u_3, \varphi_0 \vdash_Q c_1 : \varphi_1'\\ u_2:\mathtt{b}, u_3:\mathtt{b}, i:\mathtt{b}; \Theta; \Sigma; \Gamma^L; \cdot; \Delta, u_0:\mathtt{b}, u_1:\mathtt{b}, x:\tau, u_2 \le u_3, \varphi_0, \varphi_1 \vdash_Q c_2 : \varphi_2\\ \Theta; \Sigma; \Gamma^L, u_0:\mathtt{b}, u_3:\mathtt{b}, i:\mathtt{b}; \Gamma; \Delta \vdash \varphi_0' \Rightarrow \varphi \ \mathsf{true}\\ \Theta; \Sigma; \Gamma^L, u_0:\mathtt{b}, u_3:\mathtt{b}, i:\mathtt{b}; \Gamma, u_1{:}\mathtt{b}; \Delta \vdash (\varphi_0 \wedge \varphi_1') \Rightarrow \varphi \ \mathsf{true}\\ \Theta; \Sigma; \Gamma^L, u_0:\mathtt{b}, u_3:\mathtt{b}, i:\mathtt{b}; \Gamma, u_1{:}\mathtt{b}, u_2{:}\mathtt{b}; \Delta \vdash (\varphi_0 \wedge \varphi_1 \wedge \varphi_2) \Rightarrow \varphi \ \mathsf{true}\end{array}}{u_0:\mathtt{b}, u_3:\mathtt{b}, i:\mathtt{b}; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_Q (c_1; c_2) : \varphi}\ \textsc{SeqCIComp}$$

Figure 10. Sequential composition

$\mathcal{RF}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u} =$
$\{(k,c) \mid \forall e, (k,e) \in \mathcal{RE}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u} \Longrightarrow$
$(k, c\ e) \in \mathcal{RC}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}\}$

Semantics for invariant indexed types Figure 11 summaries 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.

$\mathcal{RC}(u_b.u_e.i.\varphi_1)[\![x{:}\tau.\varphi]\!]_{\theta;\mathcal{T};u_1,u_2,i} = \{(k,c) \mid$
j_b is the length of the trace from time u_1 to the end of $\mathcal{T}$
j_e is the length of the trace from time u_2 to the end of $\mathcal{T}$
$k \ge j_b > j_e$,
the configuration at time u_1 is $\xrightarrow{u_1} \sigma_b \triangleright \cdots, \langle \iota; x.c' :: K; c\rangle \cdots$
the configuration at time u_2 is $\xrightarrow{u_2} \sigma_e \triangleright \cdots, \langle \iota; K; c'[e'/x]\rangle \cdots$
between u_1 and u_2, the stack of thread i always contains $x.c'{::}K$
$\Longrightarrow (j_e, e') \in \mathcal{RE}(u_b.u_e.i.\varphi_1)[\![\tau]\!]_{\theta;\mathcal{T};u_2}$
and $\mathcal{T} \vDash \varphi[e'/x]\}$

$\mathcal{RC}(\_)[\![\varphi]\!]_{\theta;\mathcal{T};u_1,u_2,i} = \{(k,c) \mid$
j_b is the length of the trace from time u_1 to the end of $\mathcal{T}$,
j_e is the length of the trace from time u_2 to the end of $\mathcal{T}$
$k \ge j_b \ge j_e$,
the configuration at time u_1 is $\xrightarrow{u_1} \sigma_b \triangleright \cdots, \langle \iota; x.c' :: K; c\rangle \cdots$
between u_1 and u_2 (inclusive), the stack of thread i always contains prefix $x.c'{::}K$
$\Longrightarrow \mathcal{T} \vDash \varphi\}$

$\mathcal{RF}(u_b.u_e.i.\varphi_1)[\![\Pi x{:}\tau.u_1.u_2.i.(y{:}\tau'.\varphi, \varphi')]\!]_{\theta;\mathcal{T};u} =$
$\{(k,c) \mid \forall e, \forall u', u_B, u_E, \iota, u \le u' \le u_B \le u_E,$
let $\gamma = [u_B, u_E, \iota/u_1, u_2, i]$
$(k,e) \in \mathcal{RE}(u_b.u_e.i.\varphi_1)[\![\tau\gamma]\!]_{\theta;\mathcal{T};u'} \Longrightarrow$
$(k, c\ e) \in \mathcal{RC}(u_b.u_e.i.\varphi_1)[\![(y{:}\tau'\gamma.\varphi\gamma)[e/x]]\!]_{\theta;\mathcal{T};u_B,u_E,\iota}$
$\cap \mathcal{RC}()[\![\varphi'\gamma[e/x]]\!]_{\theta;\mathcal{T};u_B,u_E,\iota}\}$

$\mathcal{RA}(u_b.u_e.i.\varphi)[\![\mathtt{Act}(u_1.u_2.i.(x{:}\tau.\varphi_1, \varphi_2))]\!]_{\theta;\mathcal{T};u} =$
$\{(k,a) \mid \forall u_B, u_E, \iota, u \le u_B \le u_E,$
let $\gamma = [u_B, u_E, \iota/u_1, u_2, i]$
$(k, \mathtt{act}(a)) \in (\mathcal{RC}(u_b.u_e.i.\varphi)[\![x{:}\tau\gamma.\varphi_1\gamma]\!]_{\theta;\mathcal{T};u;u_B,u_E,\iota}$
$\cap \mathcal{RC}(u_b.u_e.i.\varphi)[\![\varphi_2\gamma]\!]_{\theta;\mathcal{T};u;u_B,u_E,\iota})\}$

$\mathcal{RA}(u_b.u_e.i.\varphi)[\![\Pi x{:}\tau.\alpha]\!]_{\theta;\mathcal{T};u} =$
$\{(k,a) \mid \forall e, \forall u', , u' \ge u, (k,e) \in \mathcal{RE}(u_b.u_e.i.\varphi)[\![\tau]\!]_{\theta;\mathcal{T};u'}$

$$
\begin{array}{ll}
\mathcal{RV}(u_b.u_e.i.\varphi)[\![\mathtt{any}]\!]_{\theta;\mathcal{T};u} & = \{(k,\mathtt{nf}) \mid k \in \mathbb{N}\} \\
\mathcal{RV}(u_b.u_e.i.\varphi)[\![X]\!]_{\theta;\mathcal{T};u} & = \theta(X) \\
\mathcal{RV}(u_b.u_e.i.\varphi)[\![\mathtt{b}]\!]_{\theta;\mathcal{T};u} & = \{(k,e) \mid (k,e) \in \mathcal{RV}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\theta;\mathcal{T};u}\} \\
\mathcal{RV}(u_b.u_e.i.\varphi)[\![\Pi x{:}\tau_1.\tau_2]\!]_{\theta;\mathcal{T};u} & = \{(k,\lambda x.e) \mid \forall j<k, \forall u', u' \geq u, \forall e', (j,e') \in \mathcal{RE}(u_b.u_e.i.\varphi)[\![\tau_1]\!]_{\theta;\mathcal{T};u'} \\
& \qquad \Longrightarrow (j, e_1[e'/x]) \in \mathcal{RE}(u_b.u_e.i.\varphi)[\![\tau_2[e'/x]]\!]_{\theta;\mathcal{T};u'}\} \cup \\
& \quad \{(k,\mathtt{nf}) \mid \mathtt{nf} \neq \lambda x.e \Longrightarrow (k,\mathtt{nf}) \in \mathcal{RE}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}\} \\
\mathcal{RV}(u_b.u_e.i.\varphi)[\![\forall X.\tau]\!]_{\theta;\mathcal{T};u} & = \{(k,\Lambda X) \mid \forall j<k, \forall \mathsf{C} \in \mathsf{Type} \Longrightarrow (j,e') \in \mathcal{RE}(u_b.u_e.i.\varphi)[\![\tau]\!]_{\theta[X\mapsto \mathsf{C}];\mathcal{T};u}\} \cup \\
& \quad \{(k,\mathtt{nf}) \mid \mathtt{nf} \neq \Lambda X.e \Longrightarrow (k,\mathtt{nf}) \in \mathcal{RE}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}\} \\
\multicolumn{2}{l}{\mathcal{RV}(u_b.u_e.i.\varphi)[\![\mathtt{comp}(u_1.u_2.i.(x{:}\tau.\varphi_1,\varphi_2))]\!]_{\theta;\mathcal{T};u} \ =} \\
\multicolumn{2}{l}{\quad \{(k,\mathtt{comp}(c)) \mid \forall u_B, u_E, \iota, u \leq u_B \leq u_E, \text{let } \gamma = [u_B, u_E, \iota/u_1, u_2, i]} \\
\multicolumn{2}{l}{\qquad\qquad (k,c) \in \mathcal{RC}(u_b.u_e.i.\varphi)[\![x{:}\tau\gamma.\varphi_1\gamma]\!]_{\theta;\mathcal{T};u_B,u_E,\iota} \cap \mathcal{RC}(\_)[\![\varphi_2\gamma]\!]_{\theta;\mathcal{T};u_B,u_E,\iota}\} \cup} \\
\multicolumn{2}{l}{\quad \{(k,\mathtt{nf}) \mid \mathtt{nf} \neq \mathtt{comp}(c) \Longrightarrow (k,\mathtt{nf}) \in \mathcal{RE}_{INV}[\![u_1.u_2.i.\varphi]\!]_{\mathcal{T};u}\}}
\end{array}
$$

$$
\mathcal{RE}(u_b.u_e.i.\varphi)[\![\tau]\!]_{\theta;\mathcal{T};u} = \{(k,e) \mid \forall j<m, e \rightarrow_\beta^m e' \not\rightarrow \Longrightarrow (k-m,e') \in \mathcal{RV}(u_b.u_e.i.\varphi)[\![\tau]\!]_{\theta;\mathcal{T};u}\}
$$

Figure 11. Semantics for inv-indexed types

$$
\Longrightarrow (k, a\ e) \in \mathcal{RA}(u_b.u_e.i.\varphi)[\![\alpha[e/x]]\!]_{\theta;\mathcal{T};u'}\}
$$

$$
\begin{array}{l}
\mathcal{RA}(u_b.u_e.i.\varphi)[\![\forall X.\alpha]\!]_{\theta;\mathcal{T};u} = \\
\{(k,a) \mid \forall j \leq k, \forall \mathsf{C} \in \mathsf{Type} \\
\qquad \Longrightarrow (j, a\ \cdot) \in \mathcal{RA}(u_b.u_e.i.\varphi)[\![\alpha]\!]_{\theta[X\mapsto \mathsf{C}];\mathcal{T};u}\}
\end{array}
$$

Formula semantics

$$
\begin{array}{ll}
[\![\mathtt{any}]\!] & = \{e \mid e \text{ is an expression}\} \\
[\![\mathtt{b}]\!] & = \{e \mid e \rightarrow^* bv\} \\
[\![\Pi x{:}\tau_1.\tau_2]\!] & = \{\lambda x.e \mid \forall e', e' \in [\![\tau_1]\!] \Longrightarrow e_1[e'/x] \in [\![\tau_2]\!]\}
\end{array}
$$

$\mathcal{T} \vDash P\ \vec{e}$	iff	$P\ \vec{e} \in \varepsilon(\mathcal{T})$
$\mathcal{T} \vDash \mathsf{start}(I,c,U)$	iff	thread I has c as the active computation with an empty stack at time U on $\mathcal{T}$
$\mathcal{T} \vDash \forall x{:}\tau.\varphi$	iff	$\forall e, e \in [\![\tau]\!]$ implies $\mathcal{T} \vDash \varphi[e/x]$
$\mathcal{T} \vDash \exists x{:}\tau.\varphi$	iff	$\exists e, e \in [\![\tau]\!]$ and $\mathcal{T} \vDash \varphi[e/x]$

E. Lemmas

Lemma 5 ($\mathcal{R}_{INV}$ is downward-closure)**.**

1. *If* $(k,c) \in \mathcal{RV}_{INV}[\![\Phi]\!]_{\mathcal{T};u}$ *then* $\forall j{<}k$, $(j,c) \in \mathcal{RV}_{INV}[\![\Phi]\!]_{\mathcal{T};u}$
2. *If* $(k,c) \in \mathcal{RE}_{INV}[\![\Phi]\!]_{\theta,\mathcal{T};u}$ *then* $\forall j{<}k$, $(j,c) \in \mathcal{RE}_{INV}[\![\Phi]\!]_{\mathcal{T};u}$
3. *If* $(k,c) \in \mathcal{RC}_{INV}[\![\Phi]\!]_{\mathcal{T};u}$ *then* $\forall j{<}k$, $(j,c) \in \mathcal{RC}_{INV}[\![\Phi]\!]_{\mathcal{T};u}$

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

Lemma 6 ($\mathcal{R}_{INV}$ is closed under delay)**.**

1. *If* $(k,e) \in \mathcal{RV}_{INV}[\![\Phi]\!]_{\mathcal{T};u}$ *then* $\forall u'{>}u$, $(k,e) \in \mathcal{RV}_{INV}[\![\Phi]\!]_{\mathcal{T};u'}$
2. *If* $(k,e) \in \mathcal{RE}_{INV}[\![\Phi]\!]_{\mathcal{T};u}$ *then* $\forall u'{>}u$, $(k,e) \in \mathcal{RE}_{INV}[\![\Phi]\!]_{\mathcal{T};u'}$
3. *If* $(k,e) \in \mathcal{RC}_{INV}[\![\Phi]\!]_{\mathcal{T};u}$ *then* $\forall u'{>}u$, $(k,e) \in \mathcal{RC}_{INV}[\![\Phi]\!]_{\mathcal{T};u'}$

Proof (sketch): By examining the definitions. □

Lemma 7 (Indexed types are confined)**.** *confine* (τ) $(u_b.u_e.i.\varphi)$ *implies*

1. $\mathcal{RV}(u_b.u_e.i.\varphi)[\![\tau]\!]_{\theta;\mathcal{T};u} = \mathcal{RV}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\theta;\mathcal{T};u}$.
2. $\mathcal{RE}(u_b.u_e.i.\varphi)[\![\tau]\!]_{\theta;\mathcal{T};u} = \mathcal{RE}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}$.
3. *for all* n, c, $(\forall u_B, u_E, I$ *s.t.* $u \leq u_B \leq u_E$, $(n,c) \in \mathcal{RC}(u_b.u_e.i.\varphi)[\![\tau.\varphi[u_B,u_E,I/u_b,u_e,i]]\!]_{\theta;\mathcal{T};u_B,u_E,I} \cap \mathcal{RC}(u_b.u_e.i.\varphi)[\![\varphi[u_B,u_E,I/u_b,u_e,i]]\!]_{\theta;\mathcal{T};u_B,u_E,I})$ *iff* $(n,c) \in \mathcal{RC}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}$

Proof. By induction on τ. 2 uses 1 directly, 1 uses 2 when τ is smaller, 3 uses 2 directly, and 1 uses 3 when τ is smaller.
Proof of 1.

case: $\tau = b$. Follows directly from the definitions
case: $\tau = \Pi x : \tau_1.\tau_2$

By assumptions
  confine (τ_1) $(u_b.u_e.i.\varphi)$ and *confine* (τ_1) $(u_b.u_e.i.\varphi)$ (1)
Assume
  $(n,\mathtt{nf}) \in \mathcal{RV}(u_b.u_e.i.\varphi)[\![\Pi x : \tau_1.\tau_2]\!]_{\theta;\mathcal{T};u}$ (2)
To show: $(n,\mathtt{nf}) \in \mathcal{RV}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}$
We first consider the case when $\mathtt{nf} = \lambda x.e_1$
Given $0 \leq j < n$, $u' \geq u$ $(j,e') \in \mathcal{RE}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u'}$
By I.H. on τ_1
  $(j,e') \in \mathcal{RE}(u_b.u_e.i.\varphi)[\![\tau_1]\!]_{\theta;\mathcal{T};u'}$
By (2)
  $(j, e_1[e'/x]) \in \mathcal{RE}(u_b.u_e.i.\varphi)[\![\tau_2[e'/x]]\!]_{\theta;\mathcal{T};u'}$ (3)
By I.H. on τ_2 and (3)
  $(j, e_1[e'/x]) \in \mathcal{RE}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u'}$ (4)
By (4)
  $(n, \lambda x.e_1) \in \mathcal{RV}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}$
Next we consider the case where $\mathtt{nf} = \Lambda X.e_1$ or $\mathtt{comp}(c)$
  this follows from the definition directly
Proofs for the other direction is similar

case: $\tau = \mathtt{comp}(u_b.u_e.i.(x{:}\tau.\varphi,\varphi))$

By assumption
  confine (τ) $(u_b.u_e.i.\varphi)$ (1)
Assume
  $(n,\mathtt{nf}) \in \mathcal{RV}(u_b.u_e.i.\varphi)[\![\mathtt{comp}(u_b.u_e.i.(x{:}\tau.\varphi,\varphi))]\!]_{\theta;\mathcal{T};u}$ (2)
To show $(n,\mathtt{nf}) \in \mathcal{RV}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}$
We show the case when $\mathtt{nf} = \mathtt{comp}(c)$, the other cases are trivial
By definitions, $\forall u_B, u_E, \iota, u \leq u_B \leq u_E$,
let $\gamma = [u_B, u_E, \iota/u_b, u_e, i]$
  $(n,c) \in \mathcal{RC}(u_b.u_e.i.\varphi)[\![x{:}\tau\gamma.\varphi_1\gamma]\!]_{\theta;\mathcal{T};u_B,u_E,\iota}$
    $\cap \mathcal{RC}(\_)[\![\varphi_2\gamma]\!]_{\theta;\mathcal{T};u_B,u_E,\iota}$ (3)
By I.H. and (3)
  $(n,c) \in \mathcal{RC}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}$ (4)
By (4)
  $(n,\mathtt{nf}) \in \mathcal{RV}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}$ (5)
The proof of the other direction is similar

3 is proven straightforwardly by expanding the definitions of the two relations.

□

Lemma 8 (Invariant confinement)**.**
φ is composable, and thread ι is silent between time u_B and u_E implies $\mathcal{T} \vDash \varphi[u_B, u_E, \iota/u_b, u_e, i]$

1. *If* $\mathtt{fa}(e) = \emptyset$, $\mathtt{fv}(e) \in \mathtt{dom}(\gamma)$, $(n,\gamma) \in \mathcal{RE}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}$ *then* $(n, e\gamma) \in \mathcal{RE}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}$
2. *If* $\mathtt{fa}(c) = \emptyset$, $\mathtt{fv}(c) \in \mathtt{dom}(\gamma)$, $(n,\gamma) \in \mathcal{RE}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}$ *then* $(n, c\gamma) \in \mathcal{RC}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}$

3. If $\mathtt{fa}(c) = \emptyset$, $\mathtt{fv}(\mathtt{fix} f(x).c) \in \mathtt{dom}(\gamma)$, $(n, \gamma) \in \mathcal{RE}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}$ *then* $(n, \mathtt{fix} f(x).c\gamma) \in \mathcal{RF}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}$

Proof. By induction on the structure of the terms. 3 needs a sub-induction on n. We show a few key cases.
Proof of 1.

case: $e = e_1\ e_2$

By I.H.
$(n, e_1\gamma) \in \mathcal{RE}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}$ (1)
$(n, e_2\gamma) \in \mathcal{RE}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}$ (2)
Assume $(e_1e_2)\gamma \to^m \mathtt{nf} \not\to$
$e_1\gamma \to^j \mathtt{nf}_1 \not\to$ (3)
We consider two cases: $\mathtt{nf}_1 = \lambda x.e$ and $\mathtt{nf}_1 \neq \lambda x.e$
Subcase $\mathtt{nf}_1 = \lambda x.e$:
By (1)
$(n - j, \lambda x.e) \in \mathcal{RV}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}$ (4)
By (2) and Lemma 5
$(n - j - 1, e_2\gamma) \in \mathcal{RE}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}$ (5)
By (4) and (5)
$(n - j - 1, e[e_2\gamma/x]) \in \mathcal{RE}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}$ (6)
By (6)
$(n, (e_1e_2)\gamma) \in \mathcal{RE}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}$ (7)
Subcase $\mathtt{nf}_1 \neq \lambda x.e$:
$(e_1e_2)\gamma \to^m \mathtt{nf}_1(e_2\gamma) \not\to$ (8)
By definitions
$(n, (e_1e_2)\gamma) \in \mathcal{RE}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}$ (9)

Proof of 3 is by sub-induction on n

case: $n = 0$
The fixpoint couldn't have returned. We only need to show that the trace satisfies φ. This is true because the thread executing the fixpoint is silent.
case: $n = k + 1$

Assume that $(k, \mathtt{fix} f(x).c\gamma) \in \mathcal{RF}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}$ (1)
To show $(k + 1, \mathtt{fix} f(x).c\gamma) \in \mathcal{RF}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}$
$\forall e, (k + 1, e) \in \mathcal{RE}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}$
To show $(k + 1, c\ e) \in \mathcal{RC}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}$
By (1),
$(k, \lambda z.\mathtt{comp}((\mathtt{fix} f(x).c\gamma)\ z)) \in \mathcal{RE}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}$ (2)
By I.H. on c and Lemma 5 and 6
$(k, c[\lambda z.\mathtt{comp}((\mathtt{fix} f(x).c\gamma)\ z)/f][e/x])$
$\in \mathcal{RC}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}$ (3)
Assume thread ι executes the fixpoint,
we consider the following time intervals:
(i)Before the fixpoint is unrolled,
(ii) the body of the fixpoint is evaluated,
(iii) the fixpoint returns e_1
By ι is silent in (i)
φ holds in (i) (4)
By (3) and φ is composable,
φ holds in (ii) and (iii)
and $(j_e, e_1) \in \mathcal{RE}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u_E}$
where u_e is the time when e_1 is returned
and j_e is the length of $\mathcal{T}$ from u_e till the end of $\mathcal{T}$ (5)
By (4) and (5)
$(k + 1, \mathtt{fix} f(x).c\gamma) \in \mathcal{RF}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};u}$

□

F. Properties of Interpretation of Types

Lemma 9. *If* $\mathtt{nf} \neq \lambda x.e$ *or* $\Lambda X.e$ *or* $\mathtt{comp}(c)$, *then* $(n, nf) \in \mathcal{RV}(\Phi)[\![\tau]\!]_{\theta;\mathcal{T};u}$
Proof (sketch): Case on τ. For all cases except when $\tau = X$, the conclusion follows from the definition of $\mathcal{RV}_{INV}[\![\Phi]\!]_{\mathcal{T};u}$.

When $\tau = X$, $\theta(X) \in \mathsf{Type}$. By the definition of Type, every $\mathsf{C} \in \mathsf{Type}$ contains all stuck terms that are not functions or suspended computations. □

Lemma 10 (Substitution)**.** *If* $\mathsf{C} = \mathcal{RV}(\Phi)[\![\tau_1]\!]_{\theta;\mathcal{T};u}$ *then*

1. $\mathcal{RV}(\Phi)[\![\tau]\!]_{\theta[X\mapsto\mathsf{C}];\mathcal{T};u} = \mathcal{RV}(\Phi)[\![\tau[\tau_1/X]]\!]_{\theta;\mathcal{T};u}$
2. $\mathcal{RE}(\Phi)[\![\tau]\!]_{\theta[X\mapsto\mathsf{C}];\mathcal{T};u} = \mathcal{RE}(\Phi)[\![\tau[\tau_1/X]]\!]_{\theta;\mathcal{T};u}$
3. $\mathcal{RC}(\Phi)[\![\eta]\!]_{\theta[X\mapsto\mathsf{C}],\mathcal{T},\Xi} = \mathcal{RC}(\Phi)[\![\eta[\tau_1/X]]\!]_{\theta,\mathcal{T},\Xi}$
4. $\mathcal{RA}(\Phi)[\![\alpha]\!]_{\theta[X\mapsto\mathsf{C}];\mathcal{T};u} = \mathcal{RA}(\Phi)[\![\alpha[\tau_1/X]]\!]_{\theta;\mathcal{T};u}$

Proof (sketch): By induction on the structure of τ, η, φ and α. □

Lemma 11 (Downward-closure)**.**

1. *If* $(k, c) \in \mathcal{RC}(\Phi)[\![\eta]\!]_{\theta,\mathcal{T},\Xi}$ *then* $\forall j<k$, $(j, c) \in \mathcal{RC}(\Phi)[\![\eta]\!]_{\theta,\mathcal{T},\Xi}$
2. *If* $\mathtt{ftv}(\tau) \subseteq \mathtt{dom}(\theta)$, $\forall X \in \mathtt{dom}(\theta)$, $\theta(X) \in \mathsf{Type}$, *and* $(k, e) \in \mathcal{RV}(\Phi)[\![\tau]\!]_{\theta;\mathcal{T};u}$, *then* $\forall j<k$, $(j, e) \in \mathcal{RV}(\Phi)[\![\tau]\!]_{\theta;\mathcal{T};u}$.
3. *If* $\mathtt{ftv}(\tau) \subseteq \mathtt{dom}(\theta)$, $\forall X \in \mathtt{dom}(\theta)$, $\theta(X) \in \mathsf{Type}$, *and* $(k, e) \in \mathcal{RE}(\Phi)[\![\tau]\!]_{\theta;\mathcal{T};u}$, *then* $\forall j<k$, $(j, e) \in \mathcal{RE}(\Phi)[\![\tau]\!]_{\theta;\mathcal{T};u}$.

Proof (sketch): By examining the definitions. Proofs of 3 uses proofs of 2 and 2 uses 1. □

Lemma 12 (Substitutions are closed under index reduction)**.**
If $\mathtt{ftv}(\Gamma) \subseteq \mathtt{dom}(\theta)$, $\forall X \in \mathtt{dom}(\theta)$, $\theta(X) \in \mathsf{Type}$, $(n, \gamma) \in \mathcal{RG}(\Phi)[\![\Gamma]\!]_{\theta;\mathcal{T};u}$, *and* $j < n$ *then* $(j, \gamma) \in \mathcal{RG}(\Phi)[\![\Gamma]\!]_{\theta;\mathcal{T};u}$.
Proof (sketch): By induction on the structure of Γ, using Lemma 11.
□

Lemma 13 (Validity of types)**.** *If* $\mathtt{ftv}(\tau) \subseteq \mathtt{dom}(\theta)$ *and* $\forall X \in \mathtt{dom}(\theta)$, $\theta(X) \in \mathsf{Type}$, *then* $\mathcal{RV}(\Phi)[\![\tau]\!]_{\theta;\mathcal{T};u} \in \mathsf{Type}$
Proof (sketch): By Lemmas 11. □

Lemma 14 (Closed under delay)**.**

1. *If* $(k, e) \in \mathcal{RV}(\Phi)[\![\tau]\!]_{\theta;\mathcal{T};u}$ *and* $u' > u$ *then* $(k, e) \in \mathcal{RV}(\Phi)[\![\tau]\!]_{\theta;\mathcal{T};u'}$.
2. *If* $(k, e) \in \mathcal{RE}(\Phi)[\![\tau]\!]_{\theta;\mathcal{T};u}$ *and* $u' > u$ *then* $(k, e) \in \mathcal{RE}(\Phi)[\![\tau]\!]_{\theta;\mathcal{T};u'}$.

Proof (sketch): By examining the definitions and use Lemma 6
□

Lemma 15 (Substitutions are closed under delay)**.** *If* $(n, \gamma) \in \mathcal{RG}[\![\Gamma]\!]_{\theta;\mathcal{T};u}\Phi$ *and* $u' > u$ *then* $(n, \gamma) \in \mathcal{RG}[\![\Gamma]\!]_{\theta;\mathcal{T};u'}\Phi$.
Proof (sketch): By induction on the structure of Γ, using Lemma 14.
□

G. Soundness

Theorem 16 (Soundness)**.** *Assume that* $\forall A :: \alpha \in \Sigma$, $\forall \Phi, \mathcal{T}, n, u, (n, A) \in \mathcal{RA}(\Phi)[\![\alpha]\!]_{\cdot;\mathcal{T};u}$, *then*

1. *(a)*
   - $\mathcal{E} :: u : \mathtt{b}; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_\Phi e : \tau$,
   - $\forall \theta \in \mathcal{RT}[\![\Theta]\!]$,
   - $\forall \gamma^L \in [\![\Gamma^L]\!]$,
   - $\forall U, U', U' \geq U$, *let* $\gamma_u = [U/u]$,
   - $\forall \mathcal{T}$, $\forall n, \gamma$, $(n; \gamma) \in \mathcal{RG}(\Phi)[\![\Gamma\gamma_u\gamma^L]\!]_{\theta;\mathcal{T};U'}$,
   - $\mathcal{T} \vDash \Delta\gamma\gamma_u\gamma^L$

   implies $(n; e\gamma) \in \mathcal{RE}(\Phi)[\![\tau\gamma\gamma_u\gamma^L]\!]_{\theta;\mathcal{T};U'}$
   (b)
   - $\mathcal{E} :: u_1, u_2, i; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_\Phi c : \eta$,
   - $\forall\ u, u_B, u_E, \iota$ *s.t.* $u \leq u_B \leq u_E$, *let* $\gamma_1 = [u_B, u_E, \iota/u_1, u_2, i]$
   - $\forall \theta \in \mathcal{RT}[\![\Theta]\!]$,
   - $\forall \gamma^L \in [\![\Gamma^L]\!]$,
   - $\forall \mathcal{T}$, $\forall n, \gamma$, $(n; \gamma) \in \mathcal{RG}(\Phi)[\![\Gamma\gamma_1\gamma^L]\!]_{\theta;\mathcal{T};u}$,
   - $\mathcal{T} \vDash \Delta\gamma\gamma_1\gamma^L$

   implies $(n; c\gamma) \in \mathcal{RC}(\Phi)[\![\eta\gamma\gamma_1\gamma^L]\!]_{\theta;\mathcal{T};u_B,u_E,\iota}$

(c) • $\mathcal{E} :: u : \mathtt{b}; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_\Phi c : \eta_c$,
• $\forall \theta \in \mathcal{RT}[\![\Theta]\!]$,
• $\forall \gamma^L \in [\![\Gamma^L]\!]$,
• $\forall U, U', U' \geq U$, *let* $\gamma_u = [U/u]$,
• $\forall \mathcal{T}$, $\forall n, \gamma$, $(n;\gamma) \in \mathcal{RG}(\Phi)[\![\Gamma\gamma_u\gamma^L]\!]_{\theta;\mathcal{T};U'}$,
• $\mathcal{T} \vDash \Delta\gamma\gamma_u\gamma^L$
implies $(n; c\gamma) \in \mathcal{RF}(\Phi)[\![\eta_c\gamma\gamma_u\gamma^L]\!]_{\theta;\mathcal{T};U'}$

(d) • $\mathcal{E} :: u : \mathtt{b}; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_\Phi a : \alpha$,
• $\forall \theta \in \mathcal{RT}[\![\Theta]\!]$,
• $\forall \gamma^L \in [\![\Gamma^L]\!]$,
• $\forall U, U', U' \geq U$, *let* $\gamma_u = [U/u]$,
• $\forall \mathcal{T}$, $\forall n, \gamma$, $(n;\gamma) \in \mathcal{RG}(\Phi)[\![\Gamma\gamma_u\gamma^L]\!]_{\theta;\mathcal{T};U'}$,
• $\mathcal{T} \vDash \Delta\gamma\gamma_u\gamma^L$
implies $(n; a\gamma) \in \mathcal{RA}(\Phi)[\![\alpha\gamma\gamma_u\gamma^L]\!]_{\theta;\mathcal{T};U'}$

2. *(a)* • $\mathcal{E} :: u : \mathtt{b}; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash e : \tau$,
• $\forall \theta \in \mathcal{RT}[\![\Theta]\!]$,
• $\forall \gamma^L \in [\![\Gamma^L]\!]$,
• $\forall U, U', U' \geq U$, *let* $\gamma_u = [U/u]$,
• $\forall \mathcal{T}$, $\forall \Phi$, $\forall n, \gamma$, $(n;\gamma) \in \mathcal{RG}(\Phi)[\![\Gamma\gamma_u\gamma^L]\!]_{\theta;\mathcal{T};U'}$,
• $\mathcal{T} \vDash \Delta\gamma\gamma_u\gamma^L$
implies $(n; e\gamma) \in \mathcal{RE}(\Phi)[\![\tau\gamma\gamma_u\gamma^L]\!]_{\theta;\mathcal{T};U'}$

(b) • $\mathcal{E} :: u_1, u_2, i; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash c : \eta$,
• $\forall\, u, u_B, u_E, \iota$ *s.t.* $u \leq u_B \leq u_E$, *let* $\gamma_1 = [u_B, u_E, \iota/u_1, u_2, i]$
• $\forall \theta \in \mathcal{RT}[\![\Theta]\!]$,
• $\forall \gamma^L \in [\![\Gamma^L]\!]$,
• $\forall \mathcal{T}$, $\forall \Phi$, $\forall n, \gamma, (n;\gamma) \in \mathcal{RG}(\Phi)[\![\Gamma\gamma_1\gamma^L]\!]_{\theta;\mathcal{T};u}$,
• $\mathcal{T} \vDash \Delta\gamma\gamma_1\gamma^L$
implies $(n; c\gamma) \in \mathcal{RC}(\Phi)[\![\eta\gamma\gamma_1\gamma^L]\!]_{\theta;\mathcal{T};u_B,u_E,\iota}$

(c) • $\mathcal{E} :: u : \mathtt{b}; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash c : \eta_c$,
• $\forall \theta \in \mathcal{RT}[\![\Theta]\!]$,
• $\forall \gamma^L \in [\![\Gamma^L]\!]$,
• $\forall U, U', U' \geq U$, *let* $\gamma_u = [U/u]$,
• $\forall \mathcal{T}$, $\forall \Phi$, $\forall n, \gamma$, $(n;\gamma) \in \mathcal{RG}(\Phi)[\![\Gamma\gamma_u\gamma^L]\!]_{\theta;\mathcal{T};U'}$,
• $\mathcal{T} \vDash \Delta\gamma\gamma_u\gamma^L$
implies $(n; c\gamma) \in \mathcal{RF}(\Phi)[\![\eta_c\gamma\gamma_u\gamma^L]\!]_{\theta;\mathcal{T};U'}$

(d) • $\mathcal{E} :: u : \mathtt{b}; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash a : \alpha$,
• $\forall \theta \in \mathcal{RT}[\![\Theta]\!]$,
• $\forall \gamma^L \in [\![\Gamma^L]\!]$,
• $\forall U, U', U' \geq U$, *let* $\gamma_u = [U/u]$,
• $\forall \mathcal{T}$, $\forall \Phi$, $\forall n, \gamma$, $(n;\gamma) \in \mathcal{RG}(\Phi)[\![\Gamma\gamma_u\gamma^L]\!]_{\theta;\mathcal{T};U'}$,
• $\mathcal{T} \vDash \Delta\gamma\gamma_u\gamma^L$
implies $(n; a\gamma) \in \mathcal{RA}(\Phi)[\![\alpha\gamma\gamma_u\gamma^L]\!]_{\theta;\mathcal{T};U'}$

(e) • $\mathcal{E} :: u_1, u_2, i; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash \varphi$ silent,
• $\forall\, u, u_B, u_E, \iota$ *s.t.* $u \leq u_B \leq u_E$,
• *let* $\gamma_1 = [u_B, u_E, \iota/u_1, u_2, i]$
• $\forall \theta \in \mathcal{RT}[\![\Theta]\!]$,
• $\forall \gamma^L \in [\![\Gamma^L]\!]$,
• $\forall \Phi$, $\forall \mathcal{T}$, $\forall n, \gamma, (n;\gamma) \in \mathcal{RG}(\Phi)[\![\Gamma\gamma_1\gamma^L]\!]_{\theta;\mathcal{T};u}$,
• j_b *is the length of* $\mathcal{T}$ *from time* u_B *to the end of* $\mathcal{T}$,
• j_e *is the length of* $\mathcal{T}$ *from time* u_E *to the end of* $\mathcal{T}$,
• $n \geq j_b \geq j_e$
• *between time* u_B *and time* u_E, *thread* ι *is silent*
• $\mathcal{T} \vDash \Delta\gamma\gamma_1$
implies $\mathcal{T} \vDash (\varphi\gamma\gamma_1)$

(f) • $\mathcal{E} :: \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash \varphi$ true,
• $\forall \theta \in \mathcal{RT}[\![\Theta]\!]$,
• $\forall \gamma^L \in [\![\Gamma^L]\!]$,
• $\forall \mathcal{T}$, $\forall \Phi$, $\forall n, \gamma, u, (n;\gamma) \in \mathcal{RG}(\Phi)[\![\Gamma\gamma^L]\!]_{\theta;\mathcal{T};u}$,
• $\mathcal{T} \vDash \Delta\gamma^L\gamma$
implies $\mathcal{T} \vDash \varphi\gamma^L\gamma$

Proof. By induction on the structure of $\mathcal{E}$.
Proof of 1.(a).

case: CONFINE

$$\frac{\varphi \text{ is trace composable} \quad \mathcal{E}' :: u_b, u_e, i; \Theta; \Sigma; \Gamma^L, u; \Gamma; \Delta \vdash \varphi \text{ silent} \quad u_b{:}\mathtt{b}, u_e{:}\mathtt{b}, i{:}\mathtt{b} \vdash \varphi \text{ ok} \quad \mathtt{fa}(e) = \emptyset \quad \mathtt{fv}(e) \subseteq \Gamma \quad \textit{confine}\ (\tau)\ (u_b.u_e.i.\varphi) \quad \textit{confine}\ (\Gamma)\ (u_b.u_e.i.\varphi)}{u; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash_{u_b.u_e.i.\varphi} e : \tau}\ \text{CONFINE}$$

By assumptions
$\theta \in \mathcal{RT}[\![\Theta]\!]$, $\forall \gamma^L \in [\![\Gamma^L]\!]$,
$\gamma_u = U/u$, $U' \geq U$, $\mathcal{T} \vDash \Delta\gamma\gamma_u\gamma^L$
and $(n;\gamma) \in \mathcal{RG}(u_b.u_e.i.\varphi)[\![\Gamma\gamma_u\gamma^L]\!]_{\theta;\mathcal{T};U'}$, (1)
By Lemma 7 and (1)
and $(n;\gamma) \in \mathcal{RE}_{INV}[\![\Phi]\!]_{\mathcal{T};U'}$ (2)
By I.H. on $\mathcal{E}'$, given any ι, u_B, and u_E,
ι is silent between u_B and u_E implies
$\mathcal{T} \vDash \varphi[u_B, u_E, \iota/u_b, u_e, i]$ (3)
By (1) and (3) and Lemma 8
$(n, e\gamma) \in \mathcal{RE}_{INV}[\![u_b.u_e.i.\varphi]\!]_{\mathcal{T};U'}$ (4)
By (4) and Lemma 7
$(n; e\gamma) \in \mathcal{RE}(u_b.u_e.i.\varphi)[\![\tau\gamma_u\gamma^L]\!]_{\theta;\mathcal{T};U'}$ (5)

Proof of 2.(a).

$$\frac{\mathcal{E}' :: \Theta; \Sigma; \Gamma^L, u, \Gamma \vdash \tau_1 \text{ ok} \quad u; \Theta; \Sigma; \Gamma^L; \Gamma, x : \tau_1; \Delta \vdash e : \tau_2}{u; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash \lambda x.e : \Pi x{:}\tau_1.\tau_2}\ \text{E-FUN}$$

case:

By assumptions
$\theta \in \mathcal{RT}[\![\Theta]\!]$, $\forall \gamma^L \in [\![\Gamma^L]\!]$,
$\gamma_u = U/u$, $U' \geq U$, $\mathcal{T} \vDash \Delta\gamma\gamma_u\gamma^L$
and $(n;\gamma) \in \mathcal{RG}(\Phi)[\![\Gamma\gamma_u\gamma^L]\!]_{\theta;\mathcal{T};U'}$, (1)
Given $j < k$, $u'' \geq U'$,
and $(j, e_0) \in \mathcal{RE}(\Phi)[\![\tau_1\gamma\gamma_u\gamma^L]\!]_{\theta;\mathcal{T};u''}$ (2)
By Lemma 12 and 15
$(j;\gamma) \in \mathcal{RG}(\Phi)[\![\Gamma\gamma_u\gamma^L]\!]_{\theta;\mathcal{T};u''}$ (3)
By (2) and (3)
$(j; \gamma[x \mapsto e_0]) \in \mathcal{RG}(\Phi)[\![(\Gamma, x : \tau_1)\gamma_u\gamma^L]\!]_{\theta;\mathcal{T};u''}$, (4)
By I.H. on $\mathcal{E}'$
$(j, e\gamma[x \mapsto e_0]) \in \mathcal{RE}(\Phi)[\![\tau_2\gamma_u\gamma^L\gamma[x \mapsto e_0]]\!]_{\theta;\mathcal{T};u''}$ (5)
By (5) is derived based on assumption in (2)
$(n, \lambda x.e\gamma) \in \mathcal{RV}(\Phi)[\![(\Pi x{:}\tau_1.\tau_2)\gamma\gamma_u\gamma^L]\!]_{\theta;\mathcal{T};U'}$ (6)
By (6)
$(n, \lambda x.e\gamma) \in \mathcal{RE}(\Phi)[\![(\Pi x{:}\tau_1.\tau_2)\gamma\gamma_u\gamma^L]\!]_{\theta;\mathcal{T};U'}$

$$\frac{\mathcal{E}_1 :: u; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash e_1 : \Pi x{:}\tau_1.\tau_2 \quad \mathcal{E}_2 :: u; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash e_2 : \tau_1}{u; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash e_1\ e_2 : \tau_2[e_2/x]}\ \text{E-APP}$$

case:

By assumptions
$\theta \in \mathcal{RT}[\![\Theta]\!]$, $\gamma^L \in [\![\Gamma^L]\!]$,
$\gamma_u = U/u$, $U' \geq U$, $\mathcal{T} \vDash \Delta\gamma\gamma_u\gamma^L$
and $(n;\gamma) \in \mathcal{RG}(\Phi)[\![\Gamma\gamma_u\gamma^L]\!]_{\theta;\mathcal{T};U'}$, (1)
By I.H. on $\mathcal{E}_2$
$(n, e_2\gamma) \in \mathcal{RE}(\Phi)[\![\tau_1\gamma\gamma_u\gamma^L]\!]_{\theta;\mathcal{T};U'}$ (2)
By I.H. on $\mathcal{E}_1$
$(n, e_1\gamma) \in \mathcal{RE}(\Phi)[\![(\Pi x{:}\tau_1.\tau_2)\gamma\gamma_u\gamma^L]\!]_{\theta;\mathcal{T};U'}$ (3)
Assume $(e_1\ e_2)\gamma \to_\beta^m \mathtt{nf} \not\to$
By (3),
$(e_1\ e_2)\gamma \to_\beta^* \mathtt{nf}_1(e_2\gamma)$,
and $(n - m, \mathtt{nf}_1) \in \mathcal{RV}(\Phi)[\![(\Pi x{:}\tau_1.\tau_2)\gamma\gamma_u\gamma^L]\!]_{\theta;\mathcal{T};U'}$ (4)
We consider two cases:
subcase 1: $\mathtt{nf}_1 = \lambda x.e_1'$
By (4)

$(n{-}m{-}1, e_1'[e_2\gamma/x]) \in \mathcal{RE}(\Phi)[\![\tau_2\gamma\gamma_u\gamma^L[e_2\gamma/x]]\!]_{\theta;\mathcal{T};U'}$ (5)
By (4) and (5)
$(n, (e_1e_2)\gamma) \in \mathcal{RE}(\Phi)[\![(\tau_2[e_2/x])\gamma\gamma_u\gamma^L]\!]_{\theta;\mathcal{T};U'}$ (6)
subcase 2: $\mathtt{nf}_1 \neq \lambda x.e_1'$
By Lemma 9
$(n-m, \mathtt{nf}_1(e_2\gamma)) \in \mathcal{RV}(\Phi)[\![\tau_2\gamma\gamma_u\gamma^L[e_2\gamma/x]]\!]_{\theta;\mathcal{T};U'}$ (8)
By (8)
$(n, (e_1e_2)\gamma) \in \mathcal{RE}(\Phi)[\![(\tau_2[e_2/x])\gamma\gamma_u\gamma^L]\!]_{\theta;\mathcal{T};U'}$ (9)

Proof of 2.(c)

case: FIX

Proof of 2.(b)

case: SEQC

$\mathcal{E}_1 :: u_0, u_1, i; \Theta; \Sigma; \Gamma^L; u_3, \Gamma; \Delta, u_0 \leq u_1 \vdash \varphi_0$ silent
$\mathcal{E}_2 :: u_1, u_2, i; \Theta; \Sigma; \Gamma^L, u_0 : \mathtt{b}, u_3; \Gamma; \Delta, u_1 \leq u_2, \varphi_0 \vdash c_1 : x{:}\tau.\varphi_1$
$\mathcal{E}_3 :: u_2, u_3, i; \Theta; \Sigma; \Gamma^L, u_0, u_1; \Gamma, x:\tau; \Delta, u_2 \leq u_3, \varphi_0, \varphi_1 \vdash c_2 : y{:}\tau'.\varphi_2$
$\mathcal{E}_4 :: \Theta; \Sigma; \Gamma^L, u_1, u_2, u_0, u_3, i; \Gamma, x{:}\tau, y : \tau'; \Delta \vdash (\varphi_0 \wedge \varphi_1 \wedge \varphi_2) \Rightarrow \varphi$ true
$\Theta; \Sigma; \Gamma^L, u_0, u_3, i; \Gamma, y : \tau' \vdash \varphi$ ok
$\mathtt{fv}(\mathtt{letc}(c_1, x.c_2)) \subseteq \mathtt{dom}(\Gamma)$

$$u_0, u_3, i; \Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash \mathtt{letc}(c_1, x.c_2) : y{:}\tau'.\varphi$$

By assumption
Pick time points u, u_B, u_E and thread id ι, s.t. $u \leq u_B \leq u_E$,
let $\gamma_1 = [u_B, u_E, \iota/u_0, u_3, i]$
Pick any trace $\mathcal{T}$, such that $\mathcal{T} \vDash \Delta\gamma^L\gamma\gamma_1$
$\theta \in \mathcal{RT}[\![\Theta]\!]$, $\gamma^L \in [\![\Gamma^L]\!]$,
$(n;\gamma) \in \mathcal{RG}(\Phi)[\![\Gamma\gamma_u\gamma^L]\!]_{\theta;\mathcal{T};U'}$, (1)
the length of the trace from time u_B to the end of $\mathcal{T}$ is j_b
the length of the trace from time u_E to the end of $\mathcal{T}$ is j_e
and $n \geq j_b \geq j_e$ (2)
the configuration at time u_B is
$\xrightarrow{u_B} \sigma_b \triangleright \cdots, \langle \iota; y.c :: K; \mathtt{lete}(e_1, x.c_2)\gamma\rangle \cdots$
the configuration at time u_E is
$\xrightarrow{u_E} \sigma_e \triangleright \cdots, \langle \iota; K; c[e/y]\rangle \cdots$
and between u_B and u_E (inclusive), the stack of thread ι always contains prefix $y.c :: K$ (3)
By the operational semantics
exists u_{m1}, u_{m2}, s.t. $u_B \leq u_{m1} \leq u_{m2} \leq u_e$
the configuration at time u_{m1} is
$\xrightarrow{u_{m1}} \sigma_{m1} \triangleright \cdots, \langle \iota; x.c_2\gamma :: y.c :: K; c_1\gamma\rangle \cdots$,
the configuration at time u_{m2} is
$\xrightarrow{u_{m2}} \sigma_{m2} \triangleright \cdots, \langle \iota; y.c :: K; c_2\gamma[e_0/x]\rangle \cdots$, (4)
By (4)
between time u_B and u_{m1}, thread ι is silent (5)
By (1),
$\mathcal{T} \vDash (\Delta\gamma^L\gamma\gamma_1, (u_0 \leq u_1)\gamma_1[u_{m1}/u_1])$ (6)
By (1)
$(j_{m1}, \gamma) \in \mathcal{RG}(\Phi)[\![\Gamma\gamma^L[u_E/u_3][u_B, u_{m1}, \iota/u_0, u_1, i]]\!]_{\theta;\mathcal{T};u}$, (7)
By I.H. on $\mathcal{E}_1$ and (5), (6) and (7)
$\mathcal{T} \vDash \varphi_0\gamma\gamma^L\gamma_1[u_{m1}/u_1]$ (8)
Let $\gamma_2 = \gamma\gamma^L\gamma_1[u_{m1}/u_1]$
By (1) and Lemma 15 and $u \leq u_{m1}$
$(j_{m1}; \gamma) \in \mathcal{RG}(\Phi)[\![\Gamma[u_B, u_{m1}, u_{m2}, u_E, \iota/u_0, u_1, u_2, u_3, i]]\!]_{\theta;\mathcal{T};u_{m1}}$ (9)
Let $\gamma_3 = [u_{m1}, u_{m2}, \iota/u_b, u_e, j]$,
By I.H. on $\mathcal{E}_2$ and (6), (8), (9)
$(n, c_1\gamma) \in \mathcal{RC}(\Phi)[\![(x{:}\tau.\varphi_1)\gamma_2\gamma_3]\!]_{\theta;\mathcal{T};u_{m1};u_{m2};\iota}$ (10)
By (10),
let j_{m2} be the length of the trace from time u_{m_2} to the end of $\mathcal{T}$
$(j_{m2}, e_0) \in \mathcal{RE}(\Phi)[\![\tau\gamma_2\gamma_3]\!]_{\theta;\mathcal{T};u_{m2}}$ and
$\mathcal{T} \vDash \varphi_1\gamma_2\gamma_3[e_0/x]$ (11)
By Lemma 12 and $j_{m2} < n$
let $\gamma_4 = \gamma\gamma^L[u_B, u_E, \iota/u_0, u_3, i][u_{m1}, u_{m2}/u_1, u_2][e_0/x]$,
$(j_{m2}, gamma[e_0/x])$
$\in \mathcal{RG}(\Phi)[\![(\Gamma, x{:}\tau)\gamma_4)[u_{m2}, u_E, \iota/u_2, u_3, i]]\!]_{\theta;\mathcal{T};u_{m2}}$ (12)
By I.H. on $\mathcal{E}_3$, (11), (12)
$(j_{m2}, c_2\gamma[e_0/x]) \in \mathcal{RC}(\Phi)[\![(y{:}\tau'.\varphi_2)\gamma_4]\!]_{\theta;\mathcal{T};u_{m2},u_E,\iota}$ (13)
By (14)
$(j_e, e) \in \mathcal{RE}(\Phi)[\![\tau'\gamma_4]\!]_{\theta;\mathcal{T};u_E}$ and
$\mathcal{T} \vDash \varphi_2\gamma_4[e/y]$ (14)
By I.H. on $\mathcal{E}_4$
$\mathcal{T} \vDash (\varphi_0 \wedge \varphi_1\gamma_e\varphi_2[e/y])\gamma_4 \Rightarrow \varphi\gamma_4[e/y]$ (15)
$\mathcal{T} \vDash \varphi\gamma_4[e/y]$ (16)
By (14) (15)
$(n, \mathtt{lete}(c_1, x.c_2)\gamma) \in \mathcal{RC}(\Phi)[\![(y : \tau'.\varphi)\gamma^L\gamma\gamma_1]\!]_{\theta;\mathcal{T};u_B,u_E,\iota}$

case: SEQCCOMP

Proof of 2.(f)

case: HONEST

$$\frac{\mathcal{E}_1 :: u_1, u_2, i; \Theta; \Sigma; \Gamma^L; \cdot; \Delta \vdash c : \varphi \quad \mathcal{E}_2 :: \Theta; \Sigma; \Gamma^L; \cdot; \Delta \vdash \mathsf{start}(I, c, u)\ \mathsf{true} \quad \Theta; \Sigma \vdash \Gamma^L, \Gamma\ \mathsf{ok}}{\Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash \forall u'{:}\mathtt{b}.(u'{>}u) \Rightarrow \varphi[u, u', I/u_1, u_2, i]\ \mathsf{true}}$$

By assumptions
$\theta \in \mathcal{RT}[\![\Theta]\!]$, $\gamma^L \in [\![\Gamma^L]\!]$,
$\mathcal{T} \vDash \Delta\gamma^L$ (1)
To show $\mathcal{T} \vDash_\theta (\forall u'.(u' > u) \Rightarrow \varphi[u, u', I/u_1, u_2, i])\gamma\gamma^L$
By I.H. on $\mathcal{E}_2$
$\mathcal{T} \vDash_\theta \mathsf{start}(I, c, u)\gamma^L$ (2)
By (2)
at time $u\gamma^L$, thread $I\gamma^L$ starts to evaluate c on an empty stack, (3)
Given any time $U' > u\gamma^L$, and k such that the length of $\mathcal{T}$ after $u\gamma^L$ is no less than k
By I.H. on $\mathcal{E}_1$
$(k, c) \in \mathcal{RC}(\Phi)[\![\varphi\gamma[u\gamma, U', I\gamma/u_1, u_2, i]]\!]_{\theta;\mathcal{T};u\gamma,U',I\gamma}$ (4)
because c starts from an empty stack,
c couldn't have returned at time U', (5)
By (4) (5) and (1) and the definition of $\mathcal{RC}$,
$\mathcal{T} \vDash_\theta \varphi\gamma^L[u\gamma, U', I\gamma/u_1, u_2, i]$ (7)

case: $\forall$I

$$\frac{\mathcal{E}' :: \Theta; \Sigma; \Gamma^L, x : \tau; \Gamma; \Delta \vdash \varphi\ \mathsf{true}}{\Theta; \Sigma; \Gamma^L; \Gamma; \Delta \vdash \forall x{:}\tau.\varphi\ \mathsf{true}}$$

By assumptions
$\theta \in \mathcal{RT}[\![\Theta]\!]$, $\gamma^L \in [\![\Gamma^L]\!]$, $\mathcal{T} \vDash \Delta\gamma\gamma^L$
and $(n;\gamma) \in \mathcal{RG}(\Phi)[\![\Gamma\gamma^L]\!]_{\theta;\mathcal{T};u}$, (1)
Given any e such that $e \in [\![\tau]\!]$
$\gamma^L[e/x] \in [\![\Gamma^L, x : \tau]\!]$ (2)
By I.H. on $\mathcal{E}'$
$\mathcal{T} \vDash \varphi\gamma^L[e/x]\gamma$ (3)
By definitions
$\mathcal{T} \vDash (\forall x{:}\tau.\varphi)\gamma^L\gamma$

□

H. Proof of State Integrity for Memoir

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

Figure 12 contains our model for the Memoir system.

Abbreviations and Definitions

Figure 13 summarizes the abbreviations we use.

Abbreviations

$$
\begin{array}{rcl}
(\varphi \wedge \psi)@u & = & (\varphi@u) \wedge (\psi@u) \\
(\varphi \vee \psi)@u & = & (\varphi@u) \vee (\psi@u) \\
(\varphi \Rightarrow \psi)@u & = & (\varphi@u) \Rightarrow (\psi@u) \\
(\neg\varphi)@u & = & \neg(\varphi@u) \\
\top@u & = & \top \\
\bot@u & = & \bot \\
(\forall x.\varphi)@u & = & \forall x.\, (\varphi@u) \\
(\exists x.\varphi)@u & = & \exists x.\, (\varphi@u) \\
(\varphi@u')@u & = & \varphi@u' \\
 & & \\
\varphi \circ (u_1, u_2) & = & \forall u.\, (u_1 < u < u_2) \Rightarrow (\varphi@u) \\
\varphi \circ (u_1, u_2] & = & \forall u.\, (u_1 < u \leq u_2) \Rightarrow (\varphi@u) \\
\varphi \circ [u_1, u_2) & = & \forall u.\, (u_1 \leq u < u_2) \Rightarrow (\varphi@u) \\
\varphi \circ [u_1, u_2] & = & \forall u.\, (u_1 \leq u \leq u_2) \Rightarrow (\varphi@u)
\end{array}
$$

Figure 13. Abbreviations

Axioms

We list the axioms we use in Figure 14.

Definitions

$$
\begin{array}{l}
\mathsf{LL}(u_1, u_2, e, j) = \mathsf{LLenter}(e, j)@u_1 \wedge \neg\mathsf{LLexit}(j) \circ [u_1, u_2) \\
\qquad\qquad\qquad\qquad \wedge \mathsf{LLexit}(j)@u_2 \\
\mathsf{InLLSess}(u, e, j) = \exists u_1.(u_1 \leq u) \wedge \mathsf{LLenter}(e, j)@u_1 \\
\qquad\qquad\qquad\qquad \wedge \neg\mathsf{LLexit}(j) \circ [u_1, u) \\
\mathsf{InSomeLLSess}(u, e) = \exists j.\mathsf{InLLSess}(u, e, j) \\
\mathsf{LLThread}(j, e) = \exists u.\mathsf{LLenter}(e, j)@u \\
\mathsf{PCRPrefix}(p, s\_hash) = \exists h.\ \mathsf{val\_pcr}(\mathtt{pcr17}, h) \wedge \mathsf{hash\_prefix}(h, s\_hash) \\
\mathsf{ExitsPCRProtected}(i, u, s\_hash) = \mathsf{LLexit}(i)@u \Rightarrow \\
\qquad \neg\mathsf{PCRPrefix}(\mathtt{pcr17}, s\_hash)@u
\end{array}
$$

Axioms

(LLChain) $\mathsf{LLChain}(\mathtt{hash\_chain}(-1, \mathtt{code\_hash}(e), ...), e)$

(LLExit)
$$
\begin{array}{l}
\forall s\_hash, u_2, e \\
\quad \mathsf{LLChain}(s\_hash, e) \Rightarrow \\
\quad\ \wedge \mathsf{val\_pcr}(\mathtt{pcr17}, s\_hash)@u_2 \\
\quad\ \wedge \neg\mathsf{InLLSess}(u_2, e) \Rightarrow \\
\quad\quad \exists j, u_3. \\
\quad\quad\quad \mathsf{LLThread}(j, e) \\
\quad\quad\quad\ \wedge \mathsf{LLexit}(j)@u_3 \\
\quad\quad\quad\ \wedge \mathsf{val\_pcr}(\mathtt{pcr17}, h)@u_3 \\
\quad\quad\quad\ \wedge \mathsf{hash\_prefix}(h, s\_hash) \\
\quad\quad\quad\ \wedge \forall u_4 \in (u_1, u_3). \\
\quad\quad\quad\quad \mathsf{val\_pcr}(\mathtt{pcr17}, s\_hash)@u \Rightarrow \mathsf{InLLSess}(u, e)
\end{array}
$$

(PCRInit) $\mathsf{val\_pcr}(p, 0)@-\infty$

(LLHonest) $\mathsf{LLEnter}(i, e)@u \Rightarrow \exists e'.\ \mathsf{start}(-\infty, e\, e', i)$

Th next two axiom schemas holds for any action $a(i, t)$

(LLAct1) $a(i, t)@u \wedge \mathsf{InSomeLLSess}(u, e) \Rightarrow \mathsf{InLLSess}(u, e, i)$

(LLAct2) $a(i, t)@u \wedge \mathsf{LLThread}(i, e) \Rightarrow \mathsf{InLLSess}(u, e, i)$

Figure 14. Definitions and Model-specific axioms about Late Launch

H.1 Proof

The proof proceeds in four stages. Each step employs the rely-guarantee technique 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 MemoirLib (runmodule).
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 is 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 service, 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.

PCR Protection. Consider an arbitrary service s. Let $s\_hash = \mathtt{hash\_chain}(-1, \mathtt{code\_hash}(runmodule), \mathtt{code\_hash}(s))$. 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,

$$\forall u.\mathsf{val\_pcr}(\mathtt{pcr17}, s\_hash)@u \Rightarrow \mathsf{InLLSess}(u, runmodule) \tag{1}$$

To prove 1, we use rely-guarantee reasoning in the style of [14]. To prove an invariant $\varphi(u)$, using rely guarantee reasoning, it is sufficient to show for a choice of $\psi(i, u)$ and $\iota(i)$ that

(1) $\varphi(-\infty)$

(2) $\forall i, u.\ (\iota(i) \wedge \forall u' < u.\ \varphi(u')) \Rightarrow \psi(u, i)$

(3)
$$
\begin{array}{l}
\quad (\varphi(u_1) \wedge \neg\varphi(u_2) \wedge (u_1 < u_2)) \Rightarrow \\
\quad \exists i, u_3.\ (u_1 < u_3 \leq u_2) \wedge \iota(i) \wedge \neg\psi(u_3, i) \wedge \\
\qquad\quad \forall u_4 \in (u_1, u_3).\ \varphi(u_4)
\end{array}
$$

We choose φ,ψ and ι as below:

$$
\begin{array}{l}
\varphi(u) = \mathsf{val\_pcr}(\mathtt{pcr17}, s\_hash)@u \Rightarrow \mathsf{InLLSess}(u, runmodule) \\
\psi(i, u) = \mathsf{ExitsPCRProtected}(i, u, s\_hash) \\
\iota(i) = \mathsf{LLThread}(i, runmodule)
\end{array}
$$

Condition (1) follows (PCRInit) and $\neg\mathsf{hash\_prefix}(0, s\_hash)$. Condition (3) follows directly from axiom (LLExit). To prove conditions (2) above, expanding out the definitions of φ, ι and ψ above, we need to show that

$$
\begin{array}{l}
\forall i, u.\ (\mathsf{LLThread}(i, runmodule) \\
\qquad \wedge \forall u' < u.\ (\mathsf{val\_pcr}(\mathtt{pcr17}, s\_hash)@u \\
\qquad\qquad \Rightarrow \mathsf{InLLSess}(u, runmodule)(u')) \\
\qquad \Rightarrow \mathsf{ExitsPCRProtected}(u, i)
\end{array} \tag{2}
$$

This can be rewritten as

$$
\begin{array}{l}
\forall i.\ (\mathsf{LLThread}(i, runmodule) \\
\qquad \wedge \forall u.\ (\forall u' < u.\ (\mathsf{val\_pcr}(\mathtt{pcr17}, s\_hash)@u' \\
\qquad\qquad \Rightarrow \mathsf{InLLSess}(u', runmodule)) \\
\qquad \Rightarrow \mathsf{ExitsPCRProtected}(u, i))
\end{array} \tag{3}
$$

Choose an arbitrary thread i such that $\mathsf{LLThread}(i, runmodule)$. Therefore, we have by (LLHonest) that for some e', $\mathsf{start}(-\infty, runmodulee', i)$. To use rule HONEST to show (3), we need to show that.

```
1  runmodule =
2   let snapshot =
3   λ(state, summary, skey).
4      enc_state ← act(encrypt skey service_state);
5      auth ← act(mac skey (enc_state, freshness_tag));
6      ret(enc_state, freshness_tag, auth)
7
8    let check_snapshot =
9   λ((enc_state, freshness_tag, auth), request, history, skey).
10     act(verify_mac skey auth);
11     freshness_tag′ ← act(hash (freshness_tag||request));
12     if(freshness_tag = history ∨ freshness_tag′ = history, act(dec skey enc_state), act(abort()))
13
14   let initialize =
15   λ(service, Nloc).
16     act(extend_pcr(pcr17, code_hash(service)));
17     act(verify_pcr(hash_chain(−1, codehash(runmodule), code_hash(service))));
18     skey ← act(gen_symkey());
19     let history_summary = 0
20     act(setNVRAMlocPerms(Nloc, pcr17));
21     act(NVRAMwrite(Nloc, (history_summary, skey)));
22     act((extend_pcr(pcr17, 0));
23     service_state ← (service ExtendPCR ResetPCR ··· ) INIT;
24     act(act(service_init(skey, service, service_state, Nloc)));
25     snap ← snapshot(service_state, history_summary, skey);
26     ret((), snap)
27
28   let execute =
29   λ(service, Nloc, snap, req).
30     act(extend_pcr(pcr17, code_hash(service)));
31     (skey, history_summary) ← act(NVRAMread Nloc);
32     service_state ← check_snapshot(snap, request, history_summary, skey);
33     new_summary ← act(hash (history_summary||req));
34     act(NVRAMwrite(Nloc, (new_summary, skey)));
35     act(extend_pcr(pcr17, 0));
36     act(service_try(skey, service, service_state, Nloc));
37     (new_state, resp) ← (service ExtendPCR ResetPCR ··· ) (EXEC(service_state, req));
38     snap ← snapshot(service_state, history_summary, skey);
39     act(act(service_invoke(skey, service, service_state, new_state, Nloc)));
40     ret(resp, snap)
41
42   λ(service, Nloc, call).
43      (resp, snap) ← (case call of
44           INIT ⇒ initialize(service, Nloc)
45         | EXEC(snap, req) ⇒ execute(service, Nloc, snap, req))
46      act(send(response, snap));
47      act(ll_exit())
```

Figure 12. `runmodule`: A model of Memoir's state isolation mechanism

$$\vdash runmodule : \Pi(s : \mathsf{any}, l : \mathtt{ptr}, snap : \mathtt{msg}).\ \mathtt{cmp}(u_b, u, i.\ (\forall u_b < u' < u(\mathsf{val\_pcr}(\mathtt{pcr17}, s\_hash)@u' \Rightarrow \mathsf{InLLSess}(u', runmodule)) \Rightarrow \mathsf{ExitsPCRProtected}(u, i))) \quad (4)$$

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 invariant:

$$f : \mathtt{cmp}(u_b, u_e, i.\ \neg\mathsf{PCRPrefix}(\mathtt{pcr17}, s\_hash)@u_b \Rightarrow \forall u \in [u_b, u_e].\ (\mathsf{InLLSession}(u, runmodule, i) \Rightarrow \neg\mathsf{PCRPrefix}(\mathtt{pcr17}, s\_hash)@u) \quad (5)$$

Therefore, we can give s the same type. We have now shown that by the end of service, the late launch session has either terminated or the value of `pcr17` is not a prefix of s_hash. Using (LLAct2), we can now show .

NVRAM Protection.

Axioms

We want to show that the permissions on the NVRAM are always tied to the value of `pcr17` being s_hash:

Axioms

$(SetPerms)$ $\mathsf{SetNVPerms}(i, Nloc, p)@u \wedge \mathsf{val\_pcr}(p, h)@u \Rightarrow \mathsf{NVPerms}(Nloc, p, h)@u$

$(GetPerms)$ $(\mathsf{SetNVPerms}(i, Nloc, p')@u \vee \mathsf{NVRead}(i, Nloc, p')@u \vee \mathsf{NVWrite}(i, Nloc, p')@u) \wedge \mathsf{NVPerms}(Nloc, p, h)@u \Rightarrow \mathsf{val\_pcr}(p, h)@u$

$(NVPerms)$ $\mathsf{NVPerms}(Nloc, p, h)@u_1 \wedge \neg\mathsf{NVPerms}(Nloc, p, h)@u_2 \wedge (u_1 < u_2) \Rightarrow \exists u_3, j.p', h'.\ (u_1 < u_3 <= u_2) \wedge \mathsf{val\_pcr}(p', h')@u_3 \wedge \mathsf{SetNVPerms}(j, Nloc, p')@u_3 \wedge (p \neq p' \vee h \neq h') \wedge \forall u_4 \in (u_1, u_3).\ \mathsf{NVPerms}(Nloc, p, h)@u_4$

Figure 15. Model-specific axioms about NVRAM

$$(\mathsf{SetNVPerms}(i, Nloc, \mathtt{pcr17}) \wedge \mathsf{val\_pcr}(\mathtt{pcr17}, s\_hash))@u_i \Rightarrow \forall (u > u_i).\ \mathsf{NVPerms}(Nloc, \mathtt{pcr17}, s\_hash)@u \tag{6}$$

Assume that for some time point u_i.

$$\mathsf{SetNVPerms}(i, Nloc, \mathtt{pcr17}) \wedge \mathsf{val\_pcr}(\mathtt{pcr17}, s\_hash))@u_i \tag{7}$$

We now need to show that

$$\forall (u > u_i) \Rightarrow \mathsf{NVPerms}(Nloc, \mathtt{pcr17}, s\_hash)@u$$

Again, we prove this invariant by rely guarantee reasoning, where we choose φ, ψ and ι to be the following.

$$\begin{aligned}
&\varphi(u) = \mathsf{NVPerms}(Nloc, \mathtt{pcr17}, s_hash)@u\\
&\psi(u, i) = (\mathsf{SetNVPerms}(i, Nloc, p) \Rightarrow (p = \mathtt{pcr17}) \wedge \mathsf{val_pcr}(\mathtt{pcr17}, s_hash))@u\\
&\iota(i) = \mathsf{LLThread}(i, runmodule)
\end{aligned}$$

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

$$\mathsf{NVPerms}(Nloc, \mathtt{pcr17}, s\_hash)@u_i$$

This holds by Axiom (SetPerms) and 7.

Expanding condition (2), choose i such that $\mathsf{LLThread}(i, runmodule)$. We need to show that $\forall u > u_i.(\forall u' \in (u_i, u).\ \phi(u')) \Rightarrow \psi(i, u)$

$$\begin{aligned}
&\vdash runmodule :\\
&\quad \Pi(s : \mathtt{any}, l : \mathtt{ptr}, snap : \mathtt{msg}).\ \mathtt{cmp}(u_b, u_e, i.\\
&\qquad \forall u \in (u_b, u_e] \forall u' \in [u_i, u).\\
&\qquad\quad \mathsf{NVPerms}(Nloc, \mathtt{pcr17}, s_hash)@u' \Rightarrow\\
&\qquad\quad \mathsf{SetNVPerms}(i, Nloc, p)@u \Rightarrow\\
&\qquad\quad (p = \mathtt{pcr17}) \wedge \mathsf{val_pcr}(\mathtt{pcr17}, s_hash)@u
\end{aligned} \tag{8}$$

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 the Nloc at all.

$$f : \mathtt{cmp}(u_b, u_e, i.\ \neg\mathsf{PCRPrefix}(\mathtt{pcr17}, s\_hash)@u_b \Rightarrow \forall u \in [u_b, u_e].\ (\mathsf{InLLSession}(u, runmodule, i) \Rightarrow \forall p.\ \neg\mathsf{SetNVRAMPerms}(i, Nloc, p)@u) \tag{9}$$

Condition (3) follows from (NVPerms), (GetPerms) and 1.

In particular, we can show from 6 and (GetPerms).

$$(\mathsf{SetNVPerms}(i, Nloc, \mathtt{pcr17}) \wedge \mathsf{val\_pcr}(\mathtt{pcr17}, s\_hash))@u_i \Rightarrow \forall (u > u_i).\ \mathsf{ReadNV}(I, Nloc)@u \Rightarrow \mathsf{val\_pcr}(\mathtt{pcr17}, s\_hash)@u \tag{10}$$

And by 1

Definitions

$\mathsf{NVContains}(Nloc, s) = \exists m.\mathsf{Contains}(m, s) \wedge \mathsf{val\_NV}(m, s)$

$\mathsf{Private}(s, Nloc, u) = \forall u' < u.(Send(i, m)@u \Rightarrow \neg\mathsf{Contains}(m, s) \wedge \forall Nloc'.(\mathsf{NVContains}(Nloc, s)@u' \Rightarrow (Nloc' = Nloc))$

$\mathsf{KeepsPrivate}(i, s, Nloc) = Send(i, m) \Rightarrow \neg\mathsf{Contains}(m, s) \wedge \forall Nloc'.(\mathsf{WriteNV}(Nloc', m) \wedge \mathsf{Contains}(m, s) \Rightarrow Nloc = Nloc')$

$\mathsf{NewInLL}(s, e) = \mathsf{New}(i, s)@u \Rightarrow \mathsf{InLLSess}(u, e, i)$

Axioms

(Shared) $\mathsf{LLChain}(h, e) \wedge \mathsf{NewInLL}(s, e) \wedge \forall u > u_i.\mathsf{NVPerms}(Nloc, \mathtt{pcr17}, h) \Rightarrow \forall u_1, u_2 \in (u_i, \infty] \mathsf{Private}(s, Nloc, u_1) \wedge \neg\mathsf{Private}(s, Nloc, u_2) \Rightarrow \exists i, u_3.(u_1 < u_3 <= u_2) (\mathsf{LLThread}(i, e) \neg\mathsf{KeepsPrivate}(i, s, Nloc)@u_3) \wedge \forall u \in (u_1, u_3)\mathsf{Private}(s, K, u))$

(POS) $(\mathsf{Private}(s, Nloc, u) \wedge \mathsf{Has}(i, s)@u \Rightarrow (\exists u'.(u' < u) \wedge \mathsf{New}(i, s)@u') \vee (\exists u'.(u' < u) \wedge \mathsf{ReadNV}(i, Nloc, m)@u' \wedge \mathsf{Contains}(m, s))$

(PrivateInit) $\mathsf{New}(s)@u \Rightarrow \mathsf{Private}(s, Nloc, u)$

(New3) $\mathsf{New}(i, n)@u \wedge \mathsf{New}(i', n)@u' \Rightarrow (i = i') \wedge (u = u')$

Assumption about about service_init

(Init) $\mathsf{service\_init}(i, skey, service, state, Nloc)@u_i \Rightarrow \exists u.(u < u_i) \wedge \mathsf{Start}(i, runmodule\ service\ Nloc\ \mathtt{INIT})@u$

Figure 16. Definitions and Model-specific axioms about Secrecy

$$(\mathsf{SetNVPerms}(i, Nloc, \mathtt{pcr17}) \wedge \mathsf{val\_pcr}(\mathtt{pcr17}, s\_hash))@u_i \Rightarrow \forall (u > u_i) \Rightarrow \mathsf{ReadNV}(I, Nloc)@u \Rightarrow \mathsf{InSomeLLSess}(u, runmodule) \tag{11}$$

Therefore, by (LLAct), we have that

$$(\mathsf{SetNVPerms}(i, Nloc, \mathtt{pcr17}) \wedge \mathsf{val\_pcr}(\mathtt{pcr17}, s\_hash))@u_i \Rightarrow \forall I, (u > u_i) \Rightarrow \mathsf{ReadNV}(I, Nloc)@u \Rightarrow \mathsf{InLLSess}(u, runmodule, I) \tag{12}$$

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

Key Secrecy. 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 the thread is the initialization thread itself.

$$\forall i, u_i, state, skey, Nloc\ \mathsf{service\_init}(i, skey, state, Nloc)@u_i \Rightarrow \forall j, u > u_i.\mathsf{Has}(j, skey)@u \Rightarrow (j = i) \vee \exists u', m.(u_i < u' < u) \wedge \mathsf{ReadNV}(j, Nloc, m)@u' \wedge \mathsf{Contains}(m, skey) \tag{13}$$

Fix I_i, u_i, $skey$, $service$, $Nloc$.

Assume $\mathsf{service\_init}(I_i, skey, service, state, Nloc)@u_i$ By (Init) and (HON), we have

We assume that $service$ has the following type:

$$(service\ ExtendPCR\ ResetPCR \cdots) : \Pi i : \mathtt{msg}.\ \mathtt{cmp}(u_b, u_e, i. (x : \mathtt{msg}.\neg\mathsf{Contains}(i, s) \Rightarrow \neg\mathsf{Contains}(x, s), \mathsf{KeepsSecret}(i, skey, Nloc) \circ [u_b, u_e])) \tag{14}$$

$$\begin{aligned}&\exists u_1, u_2, u_3.(u_1 < u_2 < u_3 < u_4 < u_i)\\ &\quad \mathsf{VerifyPCR}(\mathtt{pcr17}, s\_hash)@u_1\\ &\quad \mathsf{New}(skey)@u_2 \wedge\\ &\quad \mathsf{SetNVPerms}(I_i, Nloc, \mathtt{pcr17})@u_3 \wedge\\ &\quad \mathsf{NVWrite}(I_i, Nloc, (skey, h))@u_4 \wedge\\ &\quad \neg\mathsf{SetNVPerms}(I_i, Nloc, p) \circ (u_3, u_i] \wedge\\ &\quad \neg(\mathsf{Extend}(I_i, \mathtt{pcr17}, t) || \mathsf{Reset}(I_i, \mathtt{pcr17})) \circ (u_1, u_3]\\ &\quad \neg\mathsf{Send}(I_i, m) \circ (u_1, u_i]\end{aligned} \tag{15}$$

We prove this by another rely-guarantee proof, very similar to the proof of Kerberos in [14]:

$$\begin{aligned}\phi(u) &= \mathsf{Private}(skey, Nloc, u)\\ \psi(i, u) &= \mathsf{KeepsPrivate}(i, skey, Nloc)@u\\ I(i) &= \mathsf{LLThread}(i, runmodule)\end{aligned}$$

Condition (1) holds From (15) and (LLAct)

To prove condition (2) we again use the HONEST rule. However, the property required cannot be derived CONFINE. We assume that the

The key step here is the typing of the execution of s

$$(s\ ExtendPCR\ ResetPCR\ \cdots)\ (\mathtt{EXEC}(service\_state, req))$$

Here, we use the EQ rule As we know that $s = service$, from (14) we can assign

$$\begin{aligned}&(s\ ExtendPCR\ ResetPCR \cdots) : \Pi i : \mathtt{msg}.\ \mathtt{cmp}(u_b, u_e, i.\\ &\quad (x : \mathtt{msg}.\neg\mathsf{Contains}(i, s) \Rightarrow \neg\mathsf{Contains}(x, s),\\ &\quad \mathsf{KeepsSecret}(i, skey, Nloc) \circ [u_b, u_e]))\end{aligned} \tag{16}$$

Condition (3) Follows from (Shared), and (12).

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.

$$\begin{aligned}&\forall i, u_i, state, skey, Nloc\\ &\quad \mathsf{service\_init}(i, skey, state, Nloc, ...)@u_i \wedge \Rightarrow\\ &\quad\quad \forall h, state, state', j, j' u, u'.u > u_i \wedge u' > u_i \Rightarrow\\ &\quad\quad\quad \mathsf{mac}(j, skey, (state, h))@u \wedge \mathsf{mac}(j', skey, (state', h))@u' \Rightarrow\\ &\quad\quad\quad\quad (state = state')\end{aligned} \tag{17}$$

State Continuity The high level property we prove about Memoir is as follows:

$$\begin{aligned}&\forall u_i, state, state', skey, i_{init}, s_{init}\\ &\quad \mathsf{service\_init}(i_{init}, skey, service, s_{init})@u_i \Rightarrow\\ &\quad\quad \forall u > u_i.\ \mathsf{service\_invoke}(i, skey, state, state')@u \Rightarrow\\ &\quad\quad\quad \exists j, u' < u.\ ((\exists s.\mathsf{service\_invoke}(j, skey, s, state)@u'\\ &\quad\quad\quad\quad\quad \vee\ \mathsf{service\_try}(j, skey, state)@u'\\ &\quad\quad\quad\quad\quad \vee\ \mathsf{service\_init}(j, skey, state)@u')\\ &\quad\quad\quad\quad \wedge\ (\forall j'.\ \neg\mathsf{service\_invoke}(j', skey, \cdots) \circ (u', u]))\end{aligned} \tag{18}$$

Fix an i, u_i, $state$, $skey$,
Assume $\mathsf{service\_init}(i_{init}, skey, service, s_{init})@u_i$
For some $u > u_i$ assume that $\mathsf{service\_invoke}(i, skey, state, state')$.

Therefore we have $\mathsf{Has}(j, skey)@u$. By (13) we have that one of the two hold

$$\begin{aligned}&i = j \vee\\ &\exists u'.u_i < u' < u.\mathsf{ReadNV}(j, Nloc, m)@u' \wedge \mathsf{Contains}(m, skey)\end{aligned} \tag{19}$$

We analyze each case:

- **Case** i = j:

  We have from (Init) and $\mathsf{service\_init}(i, skey, service, s_{init})$ that

  $$\exists u.(u < u_i) \wedge \mathsf{Start}(i, runmodule\ service\ Nloc\ \mathtt{INIT})@u$$

  With HONEST, we can show that service_invoke does not occur on i and we have a contradiction.

- **Case** $\exists u' \in (u_i, u).\mathsf{ReadNV}(j, Nloc, m)@u' \wedge \mathsf{Contains}(m, skey)$: In this case, by (12) We have that $\mathsf{LLThread}(j, runmodule)$ Therefore, by HONEST

  $$\mathsf{ReadNVRAM}(j, (Nloc, h))@u' \tag{20}$$

  By (NVRAMRead), we have that $\exists u'' < u$ such that

  $$\begin{aligned}&\mathsf{WriteNV}(j', Nloc, m)@u'' \wedge\\ &\forall j''.\ \neg\mathsf{WriteNV}(j''.Nloc, m') \circ (u'', u']\end{aligned} \tag{21}$$

  Again, by (12): we have that

  $$\mathsf{LLThread}(j', runmodule) \tag{22}$$

  $$\mathsf{WriteNV}(j', Nloc, (skey, h))@u'' \tag{23}$$

  And by HONEST, we can derive that

  $$\mathsf{mac}(j', skey, (ENC_{skey}(state', h)) \tag{24}$$

  Also, we have two cases from 20 and HONEST from the branch at Line 12 of $runmodule$:

  - **Case** 1:

    $$\mathsf{verifyMAC}(j, skey, (ENC_{skey}(state), h) \tag{25}$$

    From 25 and (MAC), we have

    $$\mathsf{mac}(j'', skey, (ENC_{skey}(state), h)) \tag{26}$$

    By (17) on (25) and (28), we have $state' = state$ We then have from (24)

    $$\begin{aligned}&\mathsf{service\_invoke}(j', skey, s', state)@u'\\ &\vee\mathsf{service\_init}(j', skey, service, state)@u'\end{aligned} \tag{27}$$

    Also, from , we have that $\forall j''.\ \neg\mathsf{service\_invoke}(j'', \cdots)$.

  - **Case** 2:

    $$\mathsf{verifyMAC}(j, skey, (ENC_{skey}(state), h') \wedge h = H(req||h') \tag{28}$$

    This case proceeds similarly to the previous.