Certified In-lined Reference Monitoring on .NET[*]

Kevin W. Hamlen
Cornell University

Greg Morrisett
Harvard University

Fred B. Schneider
Cornell University

November 9, 2005

Abstract

MOBILE is an extension of the .NET Common Intermediate Language that permits certified In-Lined Reference Monitoring on Microsoft .NET architectures. MOBILE programs have the useful property that if they are well-typed with respect to a declared security policy, then they are guaranteed not to violate that security policy when executed. Thus, when an In-Lined Reference Monitor (IRM) is expressed in MOBILE, it can be certified by a simple type-checker to eliminate the need to trust the producer of the IRM. MOBILE thereby permits development of arbitrarily complex IRM producers without contributing that added complexity to the trusted computing base of the system.

Security policies in MOBILE are declarative, can involve potentially unbounded collections of objects allocated at runtime, and can regard finite- or infinite-length histories of security events exhibited by those objects. Our prototype implementation of MOBILE enforces properties expressed by finite-state security automata—one automaton for each security-relevant object, and can type-check MOBILE programs in the presence of exceptions, finalizers, concurrency, and non-termination. Executing MOBILE programs requires no change to existing .NET virtual machine implementations, since MOBILE programs consist of normal managed CIL code with extra typing annotations stored in .NET attributes.

1 Introduction

Language-based approaches to computer security have employed two major strategies for enforcing security policies over untrusted programs.

[*]Supported in part by AFOSR grant F49620-03-1-0156, National Science Foundation Grants 0430161 and CCF-0424422 (TRUST), ONR Grant N00014-01-1-0968, and a grant from Intel Corporation. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government.

- Low-level type systems, such as those used in Java bytecode [16], .NET CIL [5], and TAL for x86 [18], can enforce important program invariants like *memory safety* and *control safety*, which dictate that programs must access and transfer control only to certain suitable memory addresses throughout their executions.

- *Execution Monitoring* technologies such as Java and .NET stack inspection [10] [16, II.22.11], SASI [7], Polymer [1], and Naccio [8], use runtime checks to enforce temporal properties that can depend on the history of the program's execution. For example, SASI Java was used to enforce the policy that no program may access the network after it reads from a file [6]. For efficiency, execution monitors are often implemented as *In-lined Reference Monitors (IRM's)*, wherein the runtime checks are in-lined into the untrusted program itself [20].

The IRM approach is capable of enforcing a large class of powerful security policies, including ones that that cannot be enforced without runtime checks [12, 13]. But despite their power, the *rewriters* that automatically generate IRM's from untrusted programs are typically trusted components of the system. Since rewriters tend to be large and complex when efficient rewriting is required or complex security policies are to be enforced, the rewriter becomes significant addition to the system's trusted computing base.

In comparison, type systems that employ static type-checkers are less powerful in general, but the type-checkers tend to remain comparatively simple and therefore more trustworthy.

In this paper, we present MOBILE, an extension to the .NET CIL that makes it possible to automatically verify IRM's using a static type-checker. MOBILE (MOnitorable BIL with Effects) is an extension of BIL (Baby Intermediate Language) [11], a substantial fragment of managed .NET CIL that was used to develop generics for .NET [15]. MOBILE programs are CIL programs with additional typing annotations that track an abstract representation of program execution history. These typing annotations allow a type-checker to verify statically that the runtime checks in-lined into the untrusted program suffice to enforce a specified security policy. Once type-checked, the typing annotations can be erased, and the IRM can be safely executed as normal CIL code. This verification process allows a rewriter to be removed from the trusted computing base and replaced with a (simpler) type-checker. MOBILE thus leverages the power of IRM's while using the type-safety approach to keep the trusted computing base small.

Figure 1 summarizes a typical load path on a system that executes IRM's written in MOBILE. Untrusted, managed CIL code is first automatically rewritten according to a security policy, yielding an IRM written in MOBILE. The rewriting can be performed by either a code producer or

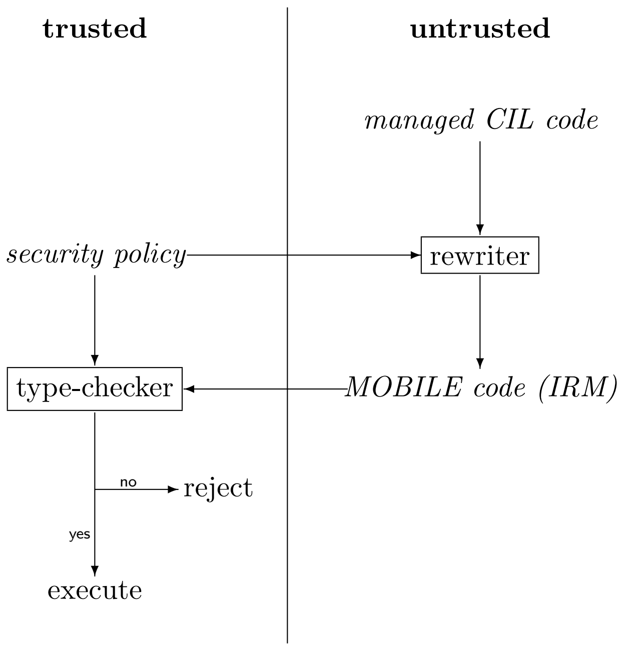

Figure 1: A MOBILE load path

by a client machine receiving the untrusted code. Since the rewriter, and therefore the IRM, remains untrusted, the IRM is then passed to a trusted type-checker that cerifies the code with respect to the original security policy. Code that satisfies the security policy will be approved by the type-checker, and is therefore safe to execute; code that is not well-typed will be rejected and would indicate a failure of the rewriter.

In this paper we focus on robust certification of MOBILE code; techniques for efficient rewriting are left to future work. Our prototype implementation of MOBILE conists of a type-checker that verifies IRM's that model security policies using finite-state security automata. The implementation can verify both single-threaded and multi-threaded managed CIL applications, and it supports language features beyond those modeled by BIL, such as exceptions and finalizers.

2 Related Work

Type-systems $\lambda_{\mathcal{A}}$ [23] and λ_{hist} [21] enforce history-based security policies over languages based on the λ-calculus. In both, program histories are tracked at the type-level using effect types that represent an abstraction of those global histories that might have been exhibited by the program prior to control reaching any given program point.

MOBILE differs from $\lambda_{\mathcal{A}}$ and λ_{hist} by tracking history on a per-object basis. That is, both $\lambda_{\mathcal{A}}$ and λ_{hist} represent a program's history as a finite or infinite sequence of global program events, where the set of all possible global program events is always finite. Policies that are only expressible using an infinite set of global program events (e.g. events parameterized by object instances) are therefore not enforceable by $\lambda_{\mathcal{A}}$ or λ_{hist}. For example, the policy that every opened file must be closed by the time the program terminates is not enforceable by either $\lambda_{\mathcal{A}}$ or λ_{hist} when the number of file objects that could be allocated during the program's execution is unbounded. In object-oriented languages like the .NET CIL, policies concerning unbounded collections of objects arise naturally, so it is not clear how $\lambda_{\mathcal{A}}$ or λ_{hist} can be extended to such settings.

MOBILE, however, enforces policies that are universally quantified over objects of any given class. For example, a MOBILE policy can dictate that, for each file handle object the program allocates, an `Open` operation must be performed on it before any `Read` operations can be performed on it. MOBILE therefore allows objects to be treated as first-class in policy specifications, whereas $\lambda_{\mathcal{A}}$ and λ_{hist} do not.

CQual [9] and Vault [4] are C-like languages that enforce history-based properties of objects by employing a flow-sensitive, object-oriented type system. Security-relevant objects in CQual or Vault programs have their base types elaborated with type qualifiers, which track the security-relevant state of the object. Control flow paths that include operations for changing the security state of an object at runtime cause the type qualifier of that object to change during type-checking. A type-checker can therefore determine if any object might enter a state at runtime that violates the security policy.

Vault's type system additionally includes variant types that allow a runtime value to reflect an object's current state. Untrusted programs can then test such values before performing security-relevant operations on the objects they track. The Vault type-checker verifies that these runtime tests are sufficient to guard against a security violation by refining an object's type qualifier along control flow paths that test such a runtime value.

Inspired by CQual and Vault, our work seeks to scale these ideas up to a large existing programming language: the managed .NET CIL. In scaling up to a larger-scale language, we adopt a somewhat different approach to tracking object security states at the type level. Both CQual and Vault assign linear types to security-relevant objects (and, in the case of Vault, to runtime state values), and use aliasing analyses to track changes to items with linear types. However, it is not clear how such analyses can be extended to support concurrency or to support an important technique commonly used by IRM's to track object security states, wherein security-relevant objects are paired with runtime values that record their states, and then such pairs are permitted to leak to the heap. Existing alias analyses cannot easily track items that are permitted to leak to the heap arbitrarily, or that can be manipulated by multiple concurrent threads of execution.

We therefore take the approach of L^3 [17], wherein linearly-typed items are permitted to leak to the heap by packing them into shared data structures. These shared object-state pairs, called *packages*, can be aliased arbitrarily and are not tracked by the type system. MOBILE provides trusted operations for packing and unpacking linear-typed items to and from shared package objects. To perform any (security-relevant) operation that might change a value with linear type, it must first be unpacked from any package that contains it. As with ownership types [3, 2], packing and unpacking operations are implemented as destructive reads, so that only one thread can perform security-relevant operations on a given security-relevant object at a time. By including appropriate pairing and unpairing operations in the code, IRM's can exploit the power of unrestricted aliasing, yet prove through the type system that all security-relevant objects are still monitored. MOBILE's type system and the CLI permissions system are both leveraged to ensure that invariants linking an object to an accurate runtime representation of its state are not violated.

Although these two pairing and unpairing operations are fixed, the presentation of MOBILE in this paper does not fix any particular method of representing object states at runtime or of dynamically testing those representations. IRM's can track object states using a variety of models including DFA's, LTL expressions, or even by recording an object's complete history at runtime. Thus, MOBILE constitutes a framework general enough to implement many different in-lining strategies used by IRM's.

3 Overview

A MOBILE *security policy* identifies a set of security-relevant object classes and assigns a set of acceptable *traces* to each such class. A trace is a finite or infinite sequence of security-relevant *events*—program operations that take a security-relevant object as an argument. A MOBILE program *satisfies* the security policy if (i) for every finite control flow path, the sequence of security-relevant events performed on every object allocated along that path is a member of the set of traces that the security policy has assigned to that object's class; and (ii) for every infinite control flow path, the sequence of security-relevant operations performed on every security-relevant object allocated along that path is a prefix of a member of the set of traces assigned to that object's class.

For example, a security policy that concerns files might identify the `System.IO.File` class provided by the .NET Common Language Runtime (CLR) as a security-relevant class, and might identify calls to the `Open`, `Read`, and `Close` methods of that class[1] as security-relevant operations. A security policy that requires programs to open files before reading them, allows at most three reads per opened file, and requires programs to close files before the program terminates, might assign $(\mathtt{O}\,(\mathtt{R} \cup \mathtt{R}^2 \cup \mathtt{R}^3)\,\mathtt{C})^\omega$ as the set of acceptable traces for class `System.IO.File` (where `O`, `R`, and `C` denote `Open`, `Read`, and `Close` events, respectively, and ω denotes finite or infinite repetition).

Although MOBILE security policies model events as operations performed on objects, *global events* that do not concern any particular object can be encoded as operations on a *global object* that is allocated at program start and destroyed at program termination. Thus, MOBILE policies can regard global events, per-object events, and combinations of the two.

For example, one might modify the example policy above by additionally requiring that at most ten reads may occur during the lifetime of the program. In that case, the global object would additionally be identified as a security-relevant object, a `Read` method call performed on any `File` object would be identified as a security-relevant event for the global object, and the global object would be assigned the set of traces denoted by $\epsilon \cup \mathtt{R} \cup \mathtt{R}^2 \cup \cdots \cup \mathtt{R}^{10}$.

We prove below that if a MOBILE program is well-typed with respect to a given security policy, then the program satisfies the security policy. That is, well-typed MOBILE programs are guaranteed, when executed, to exhibit only those sequences of events that are permitted by the security policy. This policy-adherence theorem comes in two parts. First, if a well-typed program terminates normally, then the (finite) sequence of operations exhibited on each object allocated during the program's lifetime is a trace permitted by the security policy for that object's class. Second, regardless of whether a well-typed program terminates normally, at every step in its execution, the history of security-relevant operations performed on each object is a prefix of a trace permitted by the security policy for that object's class.

A rewriter that produces IRM's from untrusted CIL code is expected to produce well-typed MOBILE code, so that the policy-adherence theorem can be used to guarantee that it is safe to execute. For this rewriting task to be feasible, MOBILE's type system must be flexible enough to permit rewriters to insert runtime security checks—well-typed code that tracks the state of security-relevant objects at runtime, testing aspects of the state that cannot be verified statically. To that end, MOBILE supports a **pack** operation that pairs a security-relevant object with a runtime value (e.g. an integer) representing an abstraction of the object's current state, and that encapsulates them into a two-field package object. MOBILE's **unpack** op-

[1]The .NET CLR's `File` class does not actually have methods with these names, but instead supports file I/O via other classes like the `StreamReader` class. We use more typical names to clarify the example.

eration can be used to unpack a package, yielding the original object that was packed along with the runtime value that represents its state. MOBILE programs can then test this runtime value to infer information about the associated object's state. Both **pack** and **unpack** are implemented as CIL method calls to a small trusted library (about ten lines of C# code).

To keep type-checking tractable, MOBILE does not permit security-relevant operations on objects that are packed. A package class' two fields are declared to be `private` so that, to access a security-relevant object directly and perform operations on it, it must first be unpacked. While unpacked, MOBILE allows only limited aliasing of security-relevant objects—none of their aliases can escape to the heap. To enforce this restriction, the **unpack** operation is implemented as a destructive read, preventing the package from being unpacked again before it is re-packed. Packages, however, are permitted to escape to the heap and to undergo unlimited aliasing. These restrictions allow the type-checker to statically track histories of unpacked objects and to ensure that packed objects are always paired with a value that accurately reflects their state. When an object is packed, it is safe for the type-checker to forget whatever information might be statically known about the object, keeping the type-checking algorithm tractable and affording the rewriter a dynamic fallback mechanism when static analysis cannot verify all security-relevant operations.

When **pack** and **unpack** are implemented as atomic operations, MOBILE can also enforce security policies in concurrent settings. In such a setting, MOBILE's type system maintains the invariant that each security-relevant object is either packed or held by at most one thread. Packed objects are always policy-adherent, while unpacked objects are tracked by the type system to ensure that they return to a policy-adherent state before they are relinquished by the thread.

4 A Formal Analysis of MOBILE

4.1 The Abstract Machine

Figure 2 gives the instruction set of MOBILE. Like BIL, MOBILE's syntax is written in postfix notation. In addition to BIL instructions[2], MOBILE includes

- instruction **evt**, which performs a security-relevant operation on an object,

[2]For simplicity, we omit BIL's value classes and managed pointers from MOBILE, but otherwise include all BIL types and instructions. Value classes and managed pointers could be included without affecting any of the results in this paper.

$I ::=$	$\textbf{ldc.i4}\ n$	integer constant
	$I_1\ I_2\ I_3\ \textbf{cond}$	conditional
	$I_1\ I_2\ \textbf{while}$	while-loop
	$I_1; I_2$	sequence
	$\textbf{ldarg}\ n$	method argument
	$I\ \textbf{starg}\ n$	store into arg
	$I_1\ \ldots\ I_n\ \textbf{newobj}\ C(\mu_1, \ldots, \mu_n)$	make new obj
	$I_0\ I_1\ \ldots\ I_n\ \textbf{callvirt}\ C{::}m.Sig$	method call
	$I\ \textbf{ldfld}\ \mu\ C{::}f$	load from field
	$I_1\ I_2\ \textbf{stfld}\ \mu\ C{::}f$	store into field
	$I\ \textbf{evt}\ e$	exhibit event
	$\textbf{newpackage}\ C$	make new package
	$I_1\ I_2\ I_3\ \textbf{pack}$	pack package
	$I\ \textbf{unpack}\ n$	unpack package
	$I_1\ I_2\ I_3\ \textbf{condst}\ C, k$	test state
	$I_1\ \ldots\ I_n\ \textbf{newhist}\ C, k$	state constructor
	$\boxed{v}$	values*
	$I\ \textbf{ret}$	method return*

*Values and return instructions do not appear in MOBILE source code, but are introduced by the small-step operational semantics as the program evaluates.

Figure 2: The MOBILE instruction set

Types	$\tau ::= \mu \mid C\langle \ell \rangle$
Untracked types	$\mu ::= \textbf{void} \mid \textbf{int32} \mid C\langle ? \rangle \mid \mathcal{R}ep_C\langle H \rangle$
Class names	C
Object identity variables	ℓ
History abstractions	$H ::= \epsilon \mid e \mid H_1 H_2 \mid H_1 \cup H_2 \mid H^\omega \mid \theta \mid H_1 \cap H_2$
History abstraction variables	θ
Method signatures	$Sig ::= \forall \Gamma_{in}.((\Psi_{in}, Fr_{in}) \multimap \exists \Gamma_{out}.(\Psi_{out}, Fr_{out}, \tau))$
Typing contexts	$\Gamma ::= \cdot \mid \Gamma,\ \ell{:}C \mid \Gamma,\ \ell{:}C\langle ? \rangle \mid \Gamma, \theta$
Object history maps	$\Psi ::= 1 \mid \Psi \star (\ell \mapsto H)$
Local variable frames	$Fr ::= (\tau_0, \ldots, \tau_n)$

Figure 3: The MOBILE type system

- instructions **newpackage** and **newhist** for creating packages and runtime state values,
- instructions **pack** and **unpack** for packing/unpacking objects and runtime state values to/from packages,
- instruction **condst**, which dynamically tests a runtime state value, and
- the pseudo-instructions $\boxed{v}$ and **ret**, which do not appear in source code but are introduced in the intermediate stages of the small-step semantics presented in §4.2.

Figure 3 provides MOBILE's type system. MOBILE types consist of void types, integers, classes, and *history abstractions* (the types of runtime state values). The type of each unpacked, security-relevant object $C\langle \ell \rangle$ is parameterized by an *object identity variable* ℓ that uniquely identifies the object. All aliases of the object have types with the same object identity variable, but other unpacked objects of the same class have types with different object identity variables. The types $C\langle ? \rangle$ of packed classes and security-irrelevant classes do not include object identity variables, and their instances are therefore not distinguishable by the type system. We consider MOBILE terms to be equivalent up to systematic alpha conversion of bound variables.

The types $\mathcal{R}ep_C\langle H \rangle$ of runtime state values are parameterized both by the class type C of the object to which they refer and by a *history abstraction*

$$\frac{}{\tau \preceq \tau}$$

$$\frac{H \subseteq H'}{\mathcal{R}ep_C\langle H\rangle \preceq \mathcal{R}ep_C\langle H'\rangle}$$

$$\frac{\tau_i \preceq \tau_i' \;\; \forall i \in 1..n}{(\tau_0, \ldots, \tau_n) \preceq (\tau_0', \ldots, \tau_n')}$$

$$\frac{Dom(\Psi) = Dom(\Psi') \qquad \Psi(\ell) \subseteq \Psi'(\ell)\ \forall \ell \in Dom(\Psi)}{\Psi \preceq \Psi'}$$

Figure 4: MOBILE subtyping

H—an ω-regular expression (plus variables and intersection) that denotes a set of traces. In such an expression, ω denotes finite or infinite repetition.

Closed (i.e. variable-less) history abstractions conform to a subset relation; we write $H_1 \subseteq H_2$ if the set of traces denoted by H_1 is a subset of the set of traces denoted by H_2. This subset relation induces a natural subtyping relation $\preceq$ given in Figure 4. Observe that the subtyping relation in Figure 4 does not recognize class subtyping of security-relevant classes. We leave support for subtyping of security-relevant classes to future work.

Type variables in MOBILE types are bound by typing contexts Γ, which assign class or package types to object identity variables ℓ and delcare any history abstraction variables θ. Object identity variables can additionally appear in object history maps Ψ, which associate a history abstraction H with each object identity variable that corresponds to an unpacked, security-relevant object. Since object identity variables uniquely identify each object instance, object history maps can be seen as a spatial conjunction ($\star$) [19] of assertions about the histories of the various unpacked objects in the heap.

A MOBILE method linearly transforms ($\multimap$) an object history map Ψ_{in} describing the security states of any unpacked security-relevant objects on the heap, and linearly transforms a frame describing any local variables on the stack Fr_{in}, to a new object history map Ψ_{out}, a new frame Fr_{out}, and a return type τ. Any new typing variables introduced by the method are bound in typing context Γ_{out}.

A complete MOBILE program consists of:

Class names	C
Field types	$\mathit{field} : (C \times f) \to \mu$
Class methods	$\mathit{methodbody} : (C{::}m.\mathit{Sig}) \to I$
Class policies	$\mathit{policy} : C \to H$

We in addition use the notation $\mathit{fields}(C)$ to refer to the number of fields in class C.

$v ::=$	result
$\mathbf{0}$	void
$i4$	integer
ℓ	heap pointer
$\boldsymbol{\mathit{rep}}_C(H)$	runtime state value
$o ::=$	heap elements
$obj_C\{f_i = v_i\}^{\overrightarrow{e}}$	object
$pkg(\ell, \boldsymbol{\mathit{rep}}_C(H))$	filled package
$pkg(\cdot)$	empty package
$h ::= \ell_i \mapsto o_i$	heap
$a ::= (v_0, \ldots, v_n)$	arguments
$s ::= (a_0, \ldots, a_n)$	stack
$\psi ::= (h, s)$	small-step store

Figure 5: The MOBILE memory model

4.2 Operational Semantics

Unlike [11], we provide a small-step operational semantics for MOBILE rather than a large-step semantics, so as to apply the policy adherence theorems presented in §4.4 to programs that do not terminate or that enter a bad state.

In MOBILE's small-step memory model, presented in Figure 5, objects consist not only of an assignment of values to fields but also a trace $\overrightarrow{e}$ that records a history of the security-relevant operations performed on the object. Although an object's field assignments are stored in memory at runtime by the virtual machine, object traces are not stored. We prove in §4.4 that it is unnecessary for the virtual machine to track and store object traces, because well-typed MOBILE code never exhibits a trace that violates the security policy.

The small-step operational semantics of MOBILE, given in Figures 6 and 7, define how a given store ψ and instruction I steps to a new store ψ' and instruction I', written $\psi, I \rightsquigarrow \psi', I'$.

Rules 17 and 18 use notation not previously defined and therefore deserve special note. Runtime operations $\text{test}_{C,k}$ and $\text{hc}_{C,k}$ test runtime state values and construct new runtime state values, respectively. Rather than fixing these two operations, we allow MOBILE to be extended with unspecified implementations of them. Different implementations of $\text{test}_{C,k}$ and $\text{hc}_{C,k}$ can therefore be used to allow MOBILE to support different collections of security policies. For example, a MOBILE system that supports security policies expressed as DFA's might implement runtime state

$$
\begin{aligned}
E ::= & [\,] \mid E\ I_2\ I_3\ \mathbf{cond} \mid E; I_2 \mid E\ \mathbf{starg}\ n \mid \\
& \boxed{v_1}\ \ldots\ \boxed{v_m}\ E\ I_1\ \ldots\ I_n\ \mathbf{newobj}\ C(\mu_1, \ldots, \mu_{m+n+1}) \mid \\
& \boxed{v_1}\ \ldots\ \boxed{v_m}\ E\ I_1\ \ldots\ I_n\ \mathbf{callvirt}\ C{::}m.\mathit{Sig} \mid E\ \mathbf{ret} \mid \\
& E\ \mathbf{ldfld}\ \mu\ C{::}f \mid E\ I_2\ \mathbf{stfld}\ \mu\ C{::}f \mid \boxed{v_1}\ E\ \mathbf{stfld}\ \mu\ C{::}f \mid \\
& E\ \mathbf{evt}\ e \mid E\ I_2\ I_3\ \mathbf{pack} \mid \boxed{v_1}\ E\ I_3\ \mathbf{pack} \mid \boxed{v_1}\ \boxed{v_2}\ E\ \mathbf{pack} \mid \\
& E\ \mathbf{unpack}\ C, k \mid E\ I_2\ I_3\ \mathbf{condst}\ C, k \mid \\
& \boxed{v_1}\ \ldots\ \boxed{v_m}\ E\ I_1\ \ldots\ I_n\ \mathbf{newhist}\ C, k
\end{aligned}
$$

Figure 6: MOBILE Evaluation Contexts

values as 32-bit integers and might support tests that compare runtime state values to integer constants (to determine which state the DFA is in). In that case, one could define for each $k \in 0..2^{32}$, $\boxed{\mathrm{hc}_{C,k}()} = \boxed{k}$ and $\mathrm{test}_{C,k}(C, \mathbf{hv}) = \{1 \text{ if } \mathbf{hv} = k, \text{ else } 0\}$. A more powerful (but more computationally expensive) MOBILE system might implement runtime state values as dynamic data structures that record an object's entire trace and might provide tests to examine such structures. In this paper, we assume only that a finite or countably infinite collection of state value constructors and tests exists and that this collection adheres to the typing constraints 19, 20, 21, and 22 presented in §4.3.

The operational semantics given in Figure 7 are for a single-threaded virtual machine without support for finalizers. To model concurrency, one could extend our stacks to consist of multiple threads, and add a small-step rule that non-deterministically chooses which thread's instruction to execute next. Finalizers could be modeled by adding another small-step rule that non-deterministically forks a finalizer thread whenever an object is unreachable. Our implementation supports concurrency and finalizers, but to simplify the presentation, we leave the analysis of these language features to future work.

4.3 Type System

The operational semantics of MOBILE presented in §4.2 permit untyped MOBILE programs to enter many bad terminal states. For example, an untyped MOBILE program might attempt to load from a non-existent field or attempt to unpack an empty package (in which case no small-step rule can be applied, and and therefore no progress can be made). We consider a terminal state to be "bad" if no progress can be made according to the small-step rules given in Figure 7 but the MOBILE program has not been reduced to a value. MOBILE's type system seeks to prevent both policy violations and bad terminal states, except that it does not seek to prevent **unpack**

$$\psi, \textbf{ldc.i4}\ i4 \rightsquigarrow \psi, \boxed{i4} \tag{1}$$

$$\frac{\psi, I \rightsquigarrow \psi', I'}{\psi, E[I] \rightsquigarrow \psi', E[I']} \tag{2}$$

$$\frac{\text{if } i4 = 0 \text{ then } j = 3 \text{ else } j = 2}{\psi, \boxed{i4}\ I_2\ I_3\ \textbf{cond} \rightsquigarrow \psi, I_j} \tag{3}$$

$$\psi, I_1\ I_2\ \textbf{while} \rightsquigarrow \psi, I_1\ (I_2; (I_1\ I_2\ \textbf{while}))\ \boxed{\mathbf{0}}\ \textbf{cond} \tag{4}$$

$$\psi, \boxed{v}; I_2 \rightsquigarrow \psi, I_2 \tag{5}$$

$$\frac{0 \le j \le n}{(h, s(v_0, \ldots, v_n)), \textbf{ldarg}\ j \rightsquigarrow (h, s(v_0, \ldots, v_n)), \boxed{v_j}} \tag{6}$$

$$\frac{0 \le j \le n}{\begin{array}{l}(h, s(v_0, \ldots, v_n)), \boxed{v}\ \textbf{starg}\ j \rightsquigarrow \\ \quad (h, s(v_0, \ldots, v_{j-1}, v, v_{j+1}, \ldots, v_n)), \boxed{\mathbf{0}}\end{array}} \tag{7}$$

$$\frac{\ell \notin \mathit{Dom}(h) \qquad n = \mathit{fields}(C)}{\begin{array}{l}(h, s), \boxed{v_1}\ \ldots\ \boxed{v_n}\ \textbf{newobj}\ C(\mu_1, \ldots, \mu_n) \rightsquigarrow \\ \qquad (h[\ell \mapsto \mathit{obj}_C\{f_i = v_i | i \in 1..n\}^{\epsilon}], s), \boxed{\ell}\end{array}} \tag{8}$$

$$\frac{\mathit{methodbody}(C{::}m.\mathit{Sig}) = I}{(h, s), \boxed{v_0}\ \ldots\ \boxed{v_n}\ \textbf{callvirt}\ C{::}m.\mathit{Sig} \rightsquigarrow (h, s(v_0, \ldots, v_n)), I\ \textbf{ret}} \tag{9}$$

$$(h, sa), \boxed{v}\ \textbf{ret} \rightsquigarrow (h, s), \boxed{v} \tag{10}$$

$$\frac{h(\ell) = \mathit{obj}_C\{\ldots, f = v, \ldots\}^{\vec{e}}}{(h, s), \boxed{\ell}\ \textbf{ldfld}\ \mu\ C{::}f \rightsquigarrow (h, s), \boxed{v}} \tag{11}$$

$$\frac{h(\ell) = \mathit{obj}_C\{\ldots, f = v, \ldots\}^{\vec{e}}}{(h, s), \boxed{\ell}\ \boxed{v'}\ \textbf{stfld}\ \mu\ C{::}f \rightsquigarrow (h[\ell \mapsto \mathit{obj}_C[f \mapsto v']], s), \boxed{\mathbf{0}}} \tag{12}$$

$$\frac{h(\ell) = \mathit{obj}_C\{\ldots\}^{\vec{e}}}{(h, s), \boxed{\ell}\ \textbf{evt}\ e_1 \rightsquigarrow (h[\ell \mapsto \mathit{obj}_C\{\ldots\}^{\vec{e}e_1}], s), \boxed{\mathbf{0}}} \tag{13}$$

$$\frac{\ell \notin \mathit{Dom}(h)}{(h, s), \textbf{newpackage}\ C \rightsquigarrow (h[\ell \mapsto \mathit{pkg}(\cdot)], s), \boxed{\ell}} \tag{14}$$

$$\frac{h(\ell) = \mathit{pkg}(\ldots)}{(h, s), \boxed{\ell}\ \boxed{\ell'}\ \boxed{\mathit{rep}_C(H)}\ \textbf{pack} \rightsquigarrow (h[\ell \mapsto \mathit{pkg}(\ell', \mathit{rep}_C(H))], s), \boxed{\mathbf{0}}} \tag{15}$$

$$\frac{h(\ell) = \mathit{pkg}(\ell', \mathit{rep}_C(H)) \qquad 0 \le j \le n}{\begin{array}{l}(h, s(v_0, \ldots, v_n)), \boxed{\ell}\ \textbf{unpack}\ j \rightsquigarrow \\ \quad \big(h[\ell \mapsto \mathit{pkg}(\cdot)], s(v_0, \ldots, v_{j-1}, \mathit{rep}_C(H), v_{j+1}, \ldots, v_n)\big), \boxed{\ell'}\end{array}} \tag{16}$$

$$\frac{\text{if } \mathrm{test}_{C,k}(\mathit{rep}_C(H)) = 0 \text{ then } j = 3 \text{ else } j = 2}{\psi, \boxed{\mathit{rep}_C(H)}\ I_2\ I_3\ \textbf{condst}\ C, k \rightsquigarrow \psi, I_j} \tag{17}$$

$$\frac{\mathit{arity}(\mathrm{hc}_{C,k}) = n}{\psi, \boxed{v_1}\ \ldots\ \boxed{v_n}\ \textbf{newhist}\ C, k \rightsquigarrow \psi, \boxed{\mathrm{hc}_{C,k}(v_1, \ldots, v_n)}} \tag{18}$$

Figure 7: Small-step Operational Sematics for MOBILE

1 (**newobj** $C()$) **starg** 1;
2 (**ldarg** 1) **evt** 9;
3 (**ldarg** 1) **evt** 8;
4 (**newpackage** C) **stfld** 2;
5 (**ldarg** 2) (**ldarg** 1) (**newhist** $C, 0$) **pack**;
6 (...) (**ldarg** 2) **stfld** ...;
7 ((**ldarg** 2) **unpack** 4) **starg** 3;
8 (**ldarg** 3) ((**ldarg** 4) **evt** 9) (...) **condst** $C, 0$

Figure 8: Sample MOBILE program

operations from being performed on empty packages. This reflects the reality that in practical settings there will always be bad terminal states that are not statically preventable. Instead, we prove in §4.4 that well-typed MOBILE programs do not commit policy violations even if they enter a bad state, such as by performing **unpack** on an empty package. Our implementation responds unpack operations on empty packages by throwing an exception that can be caught and handled by the caller. The type system prevents such an exception from leading to a policy violation.

MOBILE's type system considers a MOBILE term to be a linear operator from a history map and frame list (describing the initial heap and stack, respectively) to a new history map and frame list (describing the heap and stack yielded by the operation) along with a return type. That is, we write $\Gamma \vdash I : (\Psi; \overrightarrow{Fr}) \multimap \exists\Gamma'.(\Psi'; \overrightarrow{Fr}'; \tau')$ if term I, when evaluated in typing context Γ, takes history map Ψ and frame list $\overrightarrow{Fr}$ (in which any typing variables are bound in context Γ) to new history map Ψ' and new frame list $\overrightarrow{Fr}'$, and yields a value of type τ' (if it terminates). Any new typing variables appearing in $\overrightarrow{Fr}'$ and τ' are bound in context Γ'.

Below, we provide an informal description of MOBILE's typing rules by walking the type-checking algorithm through the sample MOBILE program given in Figure 8. A complete list of typing rules is stated formally in the appendix.

Line 1 of the sample program creates a new object of class C and stores it in local register 1. When a new security-relevant object is created, MOBILE's type system assigns it a fresh object identity variable ℓ. The return type of the the newly created object is thus $C\langle\ell\rangle$ and the new history map yielded by the operation satisfies $\Psi'(\ell) = \epsilon$; that is, new objects are initially assigned the empty trace.

As security-relevant events are performed on the object (via **evt** instructions), the type system tracks these changes by statically updating its history map to append these new events to the sequence it recorded in its history map. So for example, after processing lines 2–3 of the sample program, which perform events 9 and 8 on the object in local register 1, the type-checker's new history map would satisfy $\Psi'(\ell) = 9\,8$. At each point that a security-relevant event is performed, the type system ensures that the new trace satisfies the security policy. For example, when type-checking line 3, the type-checker would verify that $9\,8 \subseteq \mathit{pre}(\mathit{policy}(C))$, where $\mathit{policy}(C)$ denotes the set of acceptable traces assigned by the security policy to class C, and $\mathit{pre}(\mathit{policy}(C))$ denotes the set of prefixes of members of set $\mathit{policy}(C)$.

Security-relevant objects of type $C\langle\ell\rangle$ are like typical objects except that they are not permitted to escape to the heap. That is, they cannot be assigned to object fields. In order to leak a security-relevant object to the heap, a MOBILE program must first store it in a package using a **pack** instruction. This requires three steps: (1) A package must be created via a **newpackage** instruction. (2) A runtime state value must be created that accurately reflects the state of the object to be packed. This is accomplished via the **newhist** instruction, which will be described in more detail below. (3) Finally, the **pack** operation is used to store the object and the runtime state value into the package. Lines 4 and 5 of the sample program illustrate these three steps. Line 4 creates a new package and stores it in local register 2. Line 5 then fills the package using the object in local register 1 along with a newly created runtime state value.

In order for MOBILE's type system to accept a **pack** operation, it must be able to statically verify that the runtime state value is an accurate abstraction of the object being packed. That is, if the runtime state value has type $\mathcal{R}ep_C\langle H\rangle$, then the type system requires that $\Psi(\ell) \subseteq H$ where ℓ is the object identity variable of the object being packed. Additionally, since packed objects are untracked and therefore might continue to exist until the program terminates, packed objects must satisfy the security policy. That is, we require that $\Psi(\ell) \subseteq \mathit{policy}(C)$.

Packages that contain security-relevant objects can leak to the heap, as illustrated by line 6 of the sample program, which stores the package to a field of some other object.

After a **pack** operation, the type system removes object identity variable ℓ from the history map. Hence, after line 5 of the sample program, $\Psi'(\ell)$ is undefined and the object that was packed becomes inaccessible. If the program were to subsequently attempt to load from local register 1 (before replacing its contents with something else), the type-checker would reject because that register now contains a value with an invalid type. Object identity variable ℓ can therefore be thought of as a capability that has been revoked from the local scope and given to the package.

In order to perform more security-relevant events on an object, a MOBILE program must first reacquire a capability for the object by unpacking the object from its package via an **unpack** instruction. Line 7 of the sample program unpacks the package in local register 2, storing the extracted object in local register 3 and storing the runtime state value that was packaged with it in local register 4. Since packages and the objects they contain are not tracked by the type system, the type system cannot statically determine the history of a freshly unpacked object. All that is statically known is that the runtime state value that will be yielded at runtime by the **unpack** instruction will be an accurate representation of the unpacked object's history. To reflect this information statically, the type system assigns a fresh object identity variable ℓ' to the unpacked object and a fresh history variable θ to the unknown history. The unpacked object and runtime state value then have types $C\langle\ell'\rangle$ and $\mathcal{Rep}_C\langle\theta\rangle$, respectively, and the new history map satisfies $\Psi'(\ell') = \theta$.

If the sample program were at this point to perform security-relevant event k on the newly unpacked object, MOBILE's type system would reject because it would be unable to statically verify that $\theta\, k \subseteq \mathit{policy}(C)$ (since nothing is statically known about history θ). However, a MOBILE program can perform additional **evt** operations on the object by first dynamically testing the runtime state value yielded by the **unpack** operation. If a MOBILE program dynamically tests a value of type $\mathcal{Rep}_C\langle\theta\rangle$, MOBILE's type system can statically infer information about history θ within the branches of the conditional. For example, if a **condst** instruction is used to test a value with type $\mathcal{Rep}_C\langle\theta\rangle$ for equality with a value of type $\mathcal{Rep}_C\langle 9\,8\rangle$, then in the positive branch of the conditional, the type system can statically infer that $\theta = 9\,8$. If $\mathit{policy}(C) = (9\,8)^\omega$, then a MOBILE program could execute I **evt** 9 within the positive branch of such a conditional (where I is the object that was unpacked), because $9\,8\,9 \subseteq \mathit{pre}((9\,8)^\omega)$; but the type-checker would reject a program that executed I **evt** 8 in the positive branch, since $9\,8\,8 \not\subseteq \mathit{pre}((9\,8)^\omega)$.

MOBILE supports many possible schemes for representing histories at runtime and for testing them, so rather than fixing particular operations for constructing runtime state values and particular operations for testing them, we instead assume only that there exists a countable collection of constructors **newhist** C, k and conditionals **condst** C, k for all integers k, that construct runtime state values and test runtime state values (respectively) for objects of class C. We then abstractly define $HC_{C,k}(\ldots)$ to be the type $\mathcal{Rep}_C\langle H\rangle$ of a history value constructed using constructor k for security-relevant class C, and we define $\mathit{ctx}^+_{C,k}(H,\Psi)$ and $\mathit{ctx}^-_{C,k}(H,\Psi)$ to be the object history maps that refine Ψ in the positive and negative branches (respectively) of a conditional that performs test k on a history value of type $\mathcal{Rep}_C\langle H\rangle$. MOBILE supports any such refinement that is sound in the sense that

$$\mathit{test}_{C,k}(H) = 0 \Longrightarrow \Psi \preceq \mathit{ctx}^-_{C,k}(H,\Psi)(\ell) \tag{19}$$

and

$$\mathit{test}_{C,k}(H) \neq 0 \Longrightarrow \Psi \preceq \mathit{ctx}^+_{C,k}(H,\Psi)(\ell) \tag{20}$$

We further assume that each history type constructor $HC_{C,k}(\ldots)$ accurately reflects its runtime implementation, in the sense that for all history value types $\mathcal{R}ep_{C_1}\langle H_1\rangle, \ldots, \mathcal{R}ep_{C_n}\langle H_n\rangle$ such that $n = \mathit{arity}(HC_{C,k})$, there exists some H such that

$$HC_{C,k}(\mathcal{R}ep_{C_1}\langle H_1\rangle, \ldots, \mathcal{R}ep_{C_n}\langle H_n\rangle) = \mathcal{R}ep_{C}\langle H\rangle \tag{21}$$

and

$$hc_{C,k}(\mathit{rep}_{C_1}(H_1), \ldots, \mathit{rep}_{C_n}(H_n)) = \mathit{rep}_{C}(H) \tag{22}$$

In the sample program, suppose that history value constructor **newhist** $C, 0$ takes no arguments and yields a runtime value that represents history $9\,8$; and suppose that conditional test **condst** $C, 0$ compares a runtime state value to the value that represents history $9\,8$. Formally, suppose that $HC_{C,0}() = \mathcal{R}ep_{C}\langle 9\,8\rangle$ and $\mathit{ctx}^{+}_{C,0}(\theta, \Psi) = \Psi[\theta \mapsto 9\,8]$. Thus, in the positive branch of such a test, the type-checker's object history map can be refined by substituting $9\,8$ for any instances of the history variable being tested. Then if $\mathit{policy}(C) = (9\,8)^{\omega}$, a MOBILE type-checker would accept the sample program. In the positive branch of the conditional in line 8, the type-checker would infer that the object in local register 4 has history $9\,8$, and therefore it is safe to perform event 9 on it. However, if $\mathit{policy}(C) = 9\,8\,8$, then the type-checker would reject, because $9\,8\,9$ is not a prefix of $9\,8\,8$.

Our implementation of MOBILE implements history abstraction values as integers. Thus, it provides 2^{32} **newhist** operations for each security-relevant class C, defining $hc_{C,k}() = k$ for all $k \in 0..2^{32} - 1$. Tests **condst** of runtime state values are implemented as equality comparisons between the integer runtime state value to be tested and an integer constant. Thus, we define

$$\begin{aligned}
\mathrm{test}_{C,k}(\mathit{rep}_{C}(\theta)) &= \begin{cases} 1 & \text{if } \mathit{rep}_{C}(\theta) = k \\ 0 & \text{otherwise} \end{cases} \\
\mathit{ctx}^{+}_{C,k}(\theta, \Psi) &= \Psi[\theta \mapsto \theta \cap H_k] \\
\mathit{ctx}^{-}_{C,k}(\theta, \Psi) &= \Psi[\theta \mapsto \theta \cap (\cup_{i \neq k} H_i)]
\end{aligned}$$

for each integer $k \in 0..2^{32} - 1$, where H_k is a closed history abstraction statically assigned to integer constant k. The assignments of closed history abstractions H_k to integers k are not trusted, so this mapping can be defined by the MOBILE program itself (e.g., in settings where IRM's are produced by a common rewriter or where separately produced IRM's do not exchange objects) or by the policy-writer (in settings where the mapping must be defined at a system global level for consistency).

The above scheme allows a MOBILE program to represent object security states at runtime with a security automaton of 2^{32} states or less. Each state of the automaton is assigned an integer constant k, and history abstraction H_k would denote the set of traces that cause the automaton to arrive in state k.

Many other extensions to MOBILE are also possible. For example, rather than implementing runtime state values as simple integers, they could be implemented as data structures that store LTL expressions or complete trace histories. Tests of these data structures could be implemented as calls to methods in a trusted library.

4.4 Policy Adherence of MOBILE Programs

We now prove that MOBILE programs well-typed with respect to a security policy will not violate the security policy when executed. Formally, we define well-typed by

Definition 1. A method $C{::}m.Sig$ with $Sig = \forall\Gamma_{in}.(\Psi_{in}, Fr_{in}) \multimap \exists\Gamma_{out}.(\Psi_{out}, Fr_{out}, \tau)$ is *well-typed* if and only if there exists a derivation for the typing judgment $\Gamma_{in} \vdash I : (\Psi_{in}, Fr_{in}) \multimap \exists\Gamma_{out}.(\Psi_{out}, Fr_{out}, \tau)$ where $I = methodbody(C{::}m.Sig)$.

Definition 2. A MOBILE program is *well-typed* if and only if (1) for all $C{::}m.Sig \in Dom(methodbody)$, method $C{::}m.Sig$ is well-typed, and (2) there exists a method $C_{main}{::}main.Sig_{main} \in Dom(methodbody)$ with $Sig_{main} = \forall\Gamma_{in}.(\Psi_{in}, (\tau_1, \ldots, \tau_n)) \multimap \exists\Gamma_{out}.(\Psi_{out}, Fr_{out}, \tau_{out})$ such that for all substitutions $\sigma : \theta \to \overrightarrow{e}$ and all object identity variables $\ell{:}C \in (\Gamma_{in}, \Gamma_{out})$, if $\Psi_{out}(\ell) = H$ then $\sigma(H) \subseteq policy(C)$.

Part 2 of definition 2 captures the requirement that a MOBILE program's entry method must have a signature that complies with the security policy on exit.

Policy violations are defined differently depending on whether the program terminates normally. If the program terminates, MOBILE's type system guarantees that the resulting heap will be policy-adherent; whereas if the program does not terminate or enters a bad state, MOBILE guarantees only that the heap at each evaluation step will be prefix-adherent, where policy- and prefix-adherence are defined as follows:

Definition 3 (Policy Adherent)**.** A heap h is *policy-adherent* if, for all class objects $obj_C\{\ldots\}^{\overrightarrow{e}} \in Rng(h)$, $\overrightarrow{e} \subseteq policy(C)$.

Definition 4 (Prefix Adherent)**.** A heap h is *prefix-adherent* if, for all class objects $obj_C\{\ldots\}^{\overrightarrow{e}} \in Rng(h)$, $\overrightarrow{e} \subseteq pre(policy(C))$.

$$\frac{\Gamma \vdash_{heap} h : \Gamma \qquad \vdash_{hist} h : (\Gamma; \Psi) \qquad \Gamma \vdash_{stack} s : \overrightarrow{Fr}}{\Gamma \vdash (h, s) : (\Psi; \overrightarrow{Fr})} \tag{23}$$

$$\frac{\begin{array}{c}\Gamma_0 \vdash_{heap} h : \Gamma \\ \Gamma_0 \vdash \boxed{v_i} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; \mathit{field}(C, f_i)) \ \forall i \in 1..\mathit{fields}(C)\end{array}}{\Gamma_0 \vdash_{heap} h, (\ell \mapsto \mathit{obj}_C\{f_i = v_i | i \in 1..\mathit{fields}(C)\}^{\overrightarrow{e}}) : \Gamma, \ell{:}C} \tag{24}$$

$$\frac{\Gamma_0 \vdash_{heap} h : \Gamma}{\Gamma_0 \vdash_{heap} h, (\ell \mapsto \mathit{pkg}(\ldots)) : \Gamma, \ell{:}C\langle ? \rangle} \tag{25}$$

$$\frac{\Gamma_0 \vdash_{heap} h : \Gamma}{\Gamma_0 \vdash_{heap} h : \Gamma, \theta} \tag{26}$$

$$\frac{}{\Gamma_0 \vdash_{heap} \cdot : \cdot} \tag{27}$$

$$\frac{\vdash_{hist} h : (\Gamma; \Psi) \qquad \overrightarrow{e} \subseteq H}{\vdash_{hist} h, (\ell \mapsto \mathit{obj}_C\{\ldots\}^{\overrightarrow{e}}) : (\Gamma, \ell{:}C; \Psi \star (\ell \mapsto H))} \tag{28}$$

$$\frac{\vdash_{hist} h : (\Gamma; \Psi) \qquad \overrightarrow{e} \subseteq H \subseteq \mathit{policy}(C)}{\begin{array}{r}\vdash_{hist} h, (\ell \mapsto \mathit{pkg}(\ell', \boldsymbol{\mathit{rep}}_C(H))), (\ell' \mapsto \mathit{obj}_C\{\ldots\}^{\overrightarrow{e}}) : \\ (\Gamma, \ell{:}C\langle ? \rangle, \ell'{:}C; \Psi)\end{array}} \tag{29}$$

$$\frac{\vdash_{hist} h : (\Gamma; \Psi) \qquad \overrightarrow{e} \subseteq \mathit{policy}(C)}{\vdash_{hist} h, (\ell \mapsto \mathit{obj}_C\{\ldots\}^{\overrightarrow{e}}) : (\Gamma, \ell{:}C; \Psi)} \tag{30}$$

$$\frac{\vdash_{hist} h : (\Gamma; \Psi)}{\vdash_{hist} h, (\ell \mapsto \mathit{pkg}(\cdot)) : (\Gamma, \ell{:}C\langle ? \rangle; \Psi)} \tag{31}$$

$$\frac{\vdash_{hist} h : (\Gamma; \Psi)}{\vdash_{hist} h : (\Gamma, \theta; \Psi)} \tag{32}$$

$$\frac{}{\vdash_{hist} \cdot : (\cdot; 1)} \tag{33}$$

$$\frac{\Gamma \vdash_{stack} s : \overrightarrow{Fr} \qquad \Gamma \vdash \boxed{v_i} : (\Psi; \overrightarrow{Fr}_0) \multimap (\Psi; \overrightarrow{Fr}_0; \tau_i) \ \forall i \in 0..n}{\Gamma \vdash_{stack} s(v_0, \ldots, v_n) : \overrightarrow{Fr}(\tau_0, \ldots, \tau_n)} \tag{34}$$

$$\frac{}{\Gamma \vdash_{stack} \cdot : \cdot} \tag{35}$$

Figure 9: Consistency of MOBILE Statics and Dynamics

To formalize the theorem, we first define a notion of consistency between a static typing context and a runtime memory state. We say that a memory store ψ *respects* an object identity context Ψ and a list of frames $\overrightarrow{Fr}$, written $\Gamma \vdash \psi : (\Psi; \overrightarrow{Fr})$ if there exists a derivation using the the inference rules given in Figure 9. The following two theorems then establish that well-typed MOBILE programs do not violate the security policy.

Theorem 1 (Terminating Policy Adherence)**.** *Assume that a MOBILE program is well-typed, and that, as per Definition 2, its* main *method has signature* $Sig_{main} = \forall \Gamma_{in}.(\Psi_{in}, (\tau_1, \ldots, \tau_n)) \multimap \exists \Gamma_{out}.(\Psi_{out}, Fr_{out}, \tau_{out})$. *If* $\Gamma_{in} \vdash \psi : (\Psi_{in}; Fr)$ *holds and if* $\psi, methodbody(C_{main}::main.Sig) \leadsto^* (h', s'), \boxed{v}$ *holds, then* h' *is policy-adherent.*

Proof. See Appendix B. □

Theorem 2 (Non-terminating Prefix Adherence)**.** *Assume that a MOBILE program is well-typed, and assume that* $\Gamma \vdash I : (\Psi; \overrightarrow{Fr}) \multimap \exists \Gamma'.(\Psi'; \overrightarrow{Fr}'; \tau)$ *and* $\Gamma \vdash (h; s) : (\Psi; \overrightarrow{Fr})$ *hold. If* h *is prefix-adherent and* $(h, s), I \leadsto^n (h', s'), I'$ *holds, then* h' *is prefix-adherent.*

Proof. See Appendix B. □

An important consequence of both of the above theorems is that MOBILE can be implemented on existing .NET systems without modifying the memory model to store object traces at runtime. Since a static type-checker can verify that MOBILE code is well-typed, and since the theorems prove that well-typed code never exhibits a trace that violates the security policy, the runtime system need not store or monitor object traces to prevent security violations.

5 Implementation

Our prototype implementation of MOBILE consists of a type-checker for MOBILE's type system extended to the full managed subset of Microsoft's .NET CIL. The type-checker was written in Ocaml (about one thousand lines of code) and uses Microsoft's .NET ILX SDK [22] to read and manipulate .NET bytecode binaries. MOBILE programs are .NET CIL programs with typing annotations encoded as .NET method attributes. The MOBILE type-checker reads these (untrusted) annotations and verifies them in the course of type-checking. The type-checking algorithm is linear in the size of the program.

Our implementation allows security policies to identify method calls as security-relevant events. Thus, security policies can constrain the usage of resources provided by the CLR by monitoring CLR method calls and the objects they return. Our type-checker can, in principle, regard any CIL

```
struct Package {
   private object obj;
   private int state;

   public void Pack(object o, int s) {
      lock (o) { obj=o; state=s; }
   }

   public object Unpack(ref int s) {
      lock (obj) {
         object o=obj;
         if (o==null) throw new EmptyPackage();
         obj=null; s=state;
         return o;
      }
   }
}
```

Figure 10: Implementation of **pack** and **unpack**

instruction as a security-relevant event, but we leave practical investigation of this feature to future work.

Operations **pack** and **unpack** are implemented as method calls to the (very small) trusted C# library given in Figure 10. History abstraction values are implemented as integers. Thus, our **newhist** operation is simply a **ldc.i4** instruction that loads an integer constant onto the evaluation stack. Policies can statically declare for each integer constant a closed history abstraction that integer represents when used as a runtime state value at runtime. Tests of runtime state values consist of equality comparisons with integer constants in the manner described in §4.3.

With this simple support for history abstractions and tests, our type-checker can support IRM's that enforce security policies by expressing each object's state with a security automaton. Such an IRM can assign an integer constant to each state of the automaton, and can associate with each such constant a history abstraction that denotes the set of traces causing the automaton to enter the given state. The integer equality tests then allow the IRM to test whether any object's automaton is in any particular state. The type-checker must verify subset relations over the regular expressions[3]defined by these automata, which means that our type-checking

[3]Although the language of history abstractions given in Figure 3 includes infinite repetition (ω), variables (θ), and intersection ($\cap$), in practice a regular expression subset

algorithm is cubic in the size of the security policy as expressed as a security automaton.

Our type-checker also recognizes method annotations attached to finalizers of security-relevant classes. This allows security policies to be divided into two levels of enforcement: one to be enforced prior to garbage-collection and one to be enforced at garbage-collection. The part of a security policy expressed in the finalizer's precondition must be satisfied whenever the object escapes to the heap (i.e. when it is packed), since at any point after that, its package object could become orphaned and then garbage-collected. When garbage-collection occurs, the object must satisfy the secondary requirement imposed by the finalizer's postcondition. This allows an IRM to use finalizer code to enforce the security policy.

6 Conclusions and Future Work

MOBILE's type system and the theorems presented in §4.4 show that a common style of IRM, in which extra state variables and guards that model a security automaton have been in-lined into the untrusted code, can be independently verified by a type-checker, eliminating the need to trust the rewriter that produced the IRM. We verify policies that are universally quantified over unbounded collections of objects—that is, policies that require each object to exhibit a history of security-relevant events that conforms to some stated property. Properties can be expressed as DFA's, LTL expressions, or any computable language of finite and infinite event sequences.

Our implementation of MOBILE for managed Microsoft .NET CIL demonstrates that this verification procedure can be scaled to real type-safe, low-level languages. Policies can be verified in the presence of exceptions, concurrency, finalizers, and non-termination.

Our presentation of MOBILE has not addressed issues of object inheritance of security-relevant classes. Future work should examine how to safely express and implement policies that require objects related by inheritance to conform to different properties. A type-checker for such a system would need to identify when a typecast at runtime could potentially lead to a violation of the policy and provide a means for policy-adherent programs to perform necessary typecasts.

Another open problem is how to support a wider range of IRM implementations. MOBILE supports only a specific (but typical) treatment of runtime state, wherein each security-relevant object is paired with a dy-

algorithm suffices to decide subset for any history abstractions that appear in practice. This is because variables only ever appear at the beginnings of history abstractions, intersection only occurs between a variable and a closed history abstraction, and subset can be decided for ω-regular expressions lacking Kleene-stars by replacing the ω's with Kleene-stars.

namic representation of its state every time it is leaked to the heap. In some settings, it may be desirable to implement IRM's that store an object's dynamic state differently, such as in a separate array rather than packaged together with the object it models. A type system that supports these decoupled objects and states would somehow need to maintain the invariant that security-relevant objects and the runtime state values that monitor them remain consistent with one another.

We chose a type system for MOBILE that statically tracks control flow in a data-insensitive manner, with ω-regular expressions denoting sets of event sequences. This approach is appealing in that there is a natural rewriting strategy whereby well-typed MOBILE code can be automatically generated from untrusted CIL code. That is, one could replace all security-relevant objects with packages that contain those objects, and surround any security-relevant events with an unpack, a suitable state test, and a re-pack. However, a more powerful type system could employ a richer language like Hoare Logic [14] to track data-sensitive control flow. This could be potentially advantageous in that it might allow clever rewriters to eliminate some of these dynamic operations by statically proving to the type-checker that they are unnecessary. Future work should investigate rewriting strategies that could make such an approach worthwhile, and consider how to implement a type-checker that combines data-sensitive control flow analysis, spatial logic, and object-oriented languages.

Finally, not every enforceable security policy can be couched as a computable property that is universally quantified over object instances. For example, one potentially useful policy is one that requires that for every file object opened for writing, there exists an encryptor object to which its output stream has been linked. Such a policy is not supported by MOBILE because it regards both universal and existentially quantified properties that relate multiple object instances. Future work should consider how to implement IRM's that enforce such policies, and how these implementations could be type-checked so as to statically verify that the IRM satisfies the security policy.

Acknowledgments

The authors are indebted to Matthew Fluet for his helpful critiques and suggestions for this report.

References

[1] Lujo Bauer, Jay Ligatti, and David Walker. Composing security policies with polymer. In *ACM SIGPLAN Conference on Programming*

Language Design and Implementation (PLDI), pages 305–314, Chicago, Illinois, June 2005.

[2] D. G. Clarke and S. Drossopoulou. Ownership, encapsulation and disjointness of type and effect. In *Seventeenth Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 292–310, Seattle, Washington, November 2002.

[3] D. G. Clarke, J. Noble, and J. M. Potter. Simple ownership types for object containment. In *European Conference for Object-Oriented Programming (ECOOP)*, pages 53–76, Budapest, Hungary, June 2001.

[4] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 59–69, Snowbird, Utah, June 2001.

[5] ECMA. *ECMA-335: Common Language Infrastructure (CLI).* ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, second edition, December 2002.

[6] Úlfar Erlingsson and Fred B. Schneider. IRM enforcement of Java stack inspection. In *IEEE Symposium on Security and Privacy*, pages 246–255, Oakland, California, May 2000.

[7] Úlfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: A retrospective. In *WNSP: New Security Paradigms Workshop.* ACM Press, 2000.

[8] David Evans and Andrew Twynman. Flexible policy-directed code safety. In *IEEE Symposium on Security and Privacy*, pages 32–45, Oakland, California, May 1999.

[9] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, Berlin, Germany, June 2002.

[10] Li Gong. Java™ 2 platform security architecture, version 1.2. Whitepaper. © 1997–2002 Sun Microsystems, Inc.

[11] A. D. Gordon and D. Syme. Typing a multi-language intermediate code. In *Twenty-eighth ACM Symposium on Principles of Programming Languages*, pages 248–260, London, United Kingdom, January 2001.

[12] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. Computability classes for enforcement mechanisms. To appear in *ACM Transactions on Programming Languages and Systems.*

[13] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. Computability classes for enforcement mechanisms. Technical Report TR-2003-1908, Cornell University, Ithaca, New York, August 2003.

[14] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580 and 583, October 1969.

[15] Andrew Kennedy and Don Syme. The design and implementation of generics for the .NET common language runtime. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, Snowbird, Utah, June 2001.

[16] Tim Lindholm and Frank Yellin. *The Java™ Virtual Machine Specification.* Addison-Wesley, second edition, 1999.

[17] Greg Morrisett, Amal Ahmed, and Matthew Fluet. L^3: A linear language with locations. To appear in the *Seventh International Conference on Typed Lambda Calculi and Applications.*

[18] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *Proceedings of the 1999 ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, Georgia, May 1999.

[19] Peter O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Fifteenth Annual Conference of the European Association for Computer Science Logic, LNCS*, pages 1–19, Paris, France, 2001. Springer-Verlag.

[20] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and Systems Security*, 3(1):30–50, February 2000.

[21] Christian Skalka and Scott F. Smith. History effects and verification. In *Asian Programming Languages Symposium (APLAS)*, pages 107–128, November 2004.

[22] Don Syme. ILX: Extending the .NET Common IL for functional language interoperability. In Nick Benton and Andrew Kennedy, editors, *First International Workshop on Multi-Language Infrastructure and Interoperability*, volume 59.1, Florence, Italy, September 2001.

[23] David Walker. A type system for expressive security policies. In *Twenty-seventh ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 254–267, January 2000.

A Typing Rules

The following is a formal statement of MOBILE's typing rules.

$$\frac{}{\Gamma \vdash \mathbf{ldc.i4}\ n : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; \mathbf{int32})} \tag{36}$$

$$\frac{\begin{array}{c}\Gamma \vdash I_1 : (\Psi; \overrightarrow{Fr}) \multimap \exists\Gamma_1.(\Psi_1; \overrightarrow{Fr}_1; \mathbf{int32}) \\ \Gamma, \Gamma_1 \vdash I_i : (\Psi_1; \overrightarrow{Fr}_1) \multimap \exists\Gamma'.(\Psi'; \overrightarrow{Fr}'; \tau)\ \forall i \in \{2,3\}\end{array}}{\Gamma \vdash I_1\ I_2\ I_3\ \mathbf{cond} : (\Psi; \overrightarrow{Fr}) \multimap \exists\Gamma_1, \Gamma'.(\Psi'; \overrightarrow{Fr}'; \tau)} \tag{37}$$

$$\frac{\Gamma, \Gamma' \vdash I_1\ I_2\ \boxed{\mathbf{0}}\ \mathbf{cond} : (\Psi; \overrightarrow{Fr}) \multimap \exists\Gamma'.(\Psi; \overrightarrow{Fr}; \mathbf{void})}{\Gamma, \Gamma' \vdash I_1\ I_2\ \mathbf{while} : (\Psi; \overrightarrow{Fr}) \multimap \exists\Gamma'.(\Psi; \overrightarrow{Fr}; \mathbf{void})} \tag{38}$$

$$\frac{\begin{array}{c}\Gamma \vdash I_1 : (\Psi; \overrightarrow{Fr}) \multimap \exists\Gamma_1.(\Psi_1; \overrightarrow{Fr}_1; \mathbf{void}) \\ \Gamma, \Gamma_1 \vdash I_2 : (\Psi_1; \overrightarrow{Fr}_1) \multimap \exists\Gamma_2.(\Psi'; \overrightarrow{Fr}'; \tau)\end{array}}{\Gamma \vdash I_1; I_2 : (\Psi; \overrightarrow{Fr}) \multimap \exists\Gamma_1, \Gamma_2.(\Psi'; \overrightarrow{Fr}'; \tau)} \tag{39}$$

$$\frac{\begin{array}{c}\ell \in \mathit{Dom}(\Psi') \qquad \mathit{field}(C, f) = \mu \\ \Gamma \vdash I : (\Psi; \overrightarrow{Fr}) \multimap \exists\Gamma'.(\Psi'; \overrightarrow{Fr}'; C\langle\ell\rangle)\end{array}}{\Gamma \vdash I\ \mathbf{ldfld}\ \mu\ C{::}f : (\Psi; \overrightarrow{Fr}) \multimap \exists\Gamma'.(\Psi'; \overrightarrow{Fr}'; \mu)} \tag{40}$$

$$\frac{\begin{array}{cc}\Gamma \vdash I_1 : (\Psi; \overrightarrow{Fr}) \multimap \exists\Gamma_1.(\Psi_1; \overrightarrow{Fr}_1; C\langle\ell\rangle) & \ell \in \mathit{Dom}(\Psi') \\ \Gamma, \Gamma_1 \vdash I_2 : (\Psi_1; \overrightarrow{Fr}_1) \multimap \exists\Gamma_2.(\Psi'; \overrightarrow{Fr}'; \mu) & \mathit{field}(C, f) = \mu\end{array}}{\Gamma \vdash I_1\ I_2\ \mathbf{stfld}\ \mu\ C{::}f : (\Psi; \overrightarrow{Fr}) \multimap \exists\Gamma_1, \Gamma_2.(\Psi'; \overrightarrow{Fr}'; \mathbf{void})} \tag{41}$$

$$\frac{0 \leq j \leq n}{\Gamma \vdash \mathbf{ldarg}\ j : (\Psi; \overrightarrow{Fr}(\tau_0, \ldots, \tau_n)) \multimap (\Psi; \overrightarrow{Fr}(\tau_0, \ldots, \tau_n); \tau_j)} \tag{42}$$

$$\frac{\Gamma \vdash I : (\Psi; \overrightarrow{Fr}) \multimap \exists\Gamma'.(\Psi'; \overrightarrow{Fr}'(\tau_0, \ldots, \tau_n); \tau) \qquad 0 \leq j \leq n}{\begin{array}{l}\Gamma \vdash I\ \mathbf{starg}\ j : (\Psi; \overrightarrow{Fr}) \multimap \\ \qquad \exists\Gamma'.(\Psi'; \overrightarrow{Fr}'(\tau_0, \ldots, \tau_{j-1}, \tau, \tau_{j+1}, \ldots, \tau_n); \mathbf{void})\end{array}} \tag{43}$$

$$\frac{\begin{array}{l}\Gamma, \Gamma_1, \ldots, \Gamma_{i-1} \vdash I_i : (\Psi_{i-1}; \overrightarrow{Fr}_{i-1}) \multimap \\ \qquad\qquad \exists\Gamma_i.(\Psi_i; \overrightarrow{Fr}_i; \mu_i) \quad \forall i \in 1..n \\ n = \mathit{fields}(C) \quad \ell \notin \mathit{Dom}(\Gamma, \Gamma_1, \ldots, \Gamma_n) \quad \epsilon \in \mathit{pre}(\mathit{policy}(C))\end{array}}{\begin{array}{l}\Gamma \vdash I_1\ \ldots\ I_n\ \mathbf{newobj}\ C(\mu_1, \ldots, \mu_n) : \\ \quad (\Psi_0; \overrightarrow{Fr}_0) \multimap \exists\Gamma_1, \ldots, \Gamma_n, \ell{:}C.(\Psi_n \star (\ell \mapsto \epsilon); \overrightarrow{Fr}_n; C\langle\ell\rangle)\end{array}} \tag{44}$$

$$\frac{\begin{array}{c}\Gamma_0, \ldots, \Gamma_j \vdash I_j : (\Psi_j, \overrightarrow{Fr}_j) \multimap \exists\Gamma_{j+1}.(\Psi_{j+1}, \overrightarrow{Fr}_{j+1}, \tau_j)\ \forall j \in 0..n \\ \tau_0 = C\langle\ell\rangle \quad \ell \in \mathit{Dom}(\Psi_{n+1}) \quad C{::}m.\mathit{Sig} \in \mathit{Dom}(\mathit{methodbody}) \\ \Gamma_0, \ldots, \Gamma_n \vdash \mathit{Sig} <: (\Psi_{in}, (\tau_0, \ldots, \tau_n)) \multimap \exists\Gamma_{out}.(\Psi_{out}, Fr_{out}, \tau) \\ \Psi_{n+1} = \Psi_{unused} \star \Psi_{in}\end{array}}{\begin{array}{l}\Gamma_0 \vdash I_0\ \ldots\ I_n\ \mathbf{callvirt}\ C{::}m.\mathit{Sig} : \\ \quad (\Psi_0, \overrightarrow{Fr}_0) \multimap \exists\Gamma_1, \ldots, \Gamma_{n+1}, \Gamma_{out}.(\Psi_{unused} \star \Psi_{out}, \overrightarrow{Fr}_{n+1}, \tau)\end{array}} \tag{45}$$

$$\frac{\begin{array}{c}He \subseteq \mathit{pre}(\mathit{policy}(C)) \\ \Gamma \vdash I : (\Psi; \overrightarrow{Fr}) \multimap \exists\Gamma'.(\Psi' \star (\ell \mapsto H); \overrightarrow{Fr}'; C\langle\ell\rangle)\end{array}}{\Gamma \vdash I\ \mathbf{evt}\ e : (\Psi; \overrightarrow{Fr}) \multimap \exists\Gamma'.(\Psi' \star (\ell \mapsto He); \overrightarrow{Fr}'; \mathbf{void})} \tag{46}$$

$$\frac{\ell \notin \mathit{Dom}(\Gamma)}{\Gamma \vdash \mathbf{newpackage}\ C : (\Psi; \overrightarrow{Fr}) \multimap \exists\ell{:}C\langle ?\rangle.(\Psi; \overrightarrow{Fr}; C\langle ?\rangle)} \tag{47}$$

$$\frac{\begin{array}{c} H \subseteq H' \subseteq \mathit{policy}(C) \\ \Gamma \vdash I_1 : (\Psi; \overrightarrow{Fr}) \multimap \exists \Gamma_1.(\Psi_1; \overrightarrow{Fr}_1; C\langle ? \rangle) \\ \Gamma, \Gamma_1 \vdash I_2 : (\Psi_1; \overrightarrow{Fr}_1) \multimap \exists \Gamma_2.(\Psi_2; \overrightarrow{Fr}_2; C\langle \ell \rangle) \\ \Gamma, \Gamma_1, \Gamma_2 \vdash I_3 : (\Psi_2; \overrightarrow{Fr}_2) \multimap \exists \Gamma_3.(\Psi' \star (\ell \mapsto H); \overrightarrow{Fr}'; \mathcal{Rep}_C\langle H' \rangle) \end{array}}{\Gamma \vdash I_1\ I_2\ I_3\ \mathbf{pack} : (\Psi; \overrightarrow{Fr}) \multimap \exists \Gamma_1, \Gamma_2, \Gamma_3.(\Psi'; \overrightarrow{Fr}'; \mathbf{void})} \tag{48}$$

$$\frac{\begin{array}{c} \ell \notin \mathit{Dom}(\Psi') \qquad \theta \notin \mathit{Dom}(\Gamma) \\ \Gamma \vdash I : (\Psi; \overrightarrow{Fr}) \multimap \exists \Gamma'.(\Psi'; \overrightarrow{Fr}'(\tau_0, \ldots, \tau_n); C\langle ? \rangle) \end{array}}{\begin{array}{l} \Gamma \vdash I\ \mathbf{unpack}\ j : (\Psi; \overrightarrow{Fr}) \multimap \exists \Gamma', \ell{:}C, \theta. \\ \quad (\Psi', \ell \mapsto \theta; \overrightarrow{Fr}'(\tau_0, \ldots, \tau_{j-1}, \mathcal{Rep}_C\langle \theta \rangle, \tau_{j+1}, \ldots, \tau_n); C\langle \ell \rangle) \end{array}} \tag{49}$$

$$\frac{\begin{array}{c} \Gamma \vdash I_1 : (\Psi; \overrightarrow{Fr}) \multimap \exists \Gamma_1.(\Psi_1; \overrightarrow{Fr}_1; \mathcal{Rep}_C\langle H \rangle) \\ \Gamma, \Gamma_1 \vdash I_2 : (\mathit{ctx}^+_{C,k}(H, \Psi_1); \overrightarrow{Fr}_1) \multimap \exists \Gamma'.(\Psi'; \overrightarrow{Fr}'; \tau) \\ \Gamma, \Gamma_1 \vdash I_3 : (\mathit{ctx}^-_{C,k}(H, \Psi_1); \overrightarrow{Fr}_1) \multimap \exists \Gamma'.(\Psi'; \overrightarrow{Fr}'; \tau) \end{array}}{\Gamma \vdash I_1\ I_2\ I_3\ \mathbf{condst}\ k : (\Psi; \overrightarrow{Fr}) \multimap \exists \Gamma_1, \Gamma'.(\Psi'; \overrightarrow{Fr}'; \tau)} \tag{50}$$

$$\frac{\Gamma \vdash I_i : (\Psi_{i-1}; \overrightarrow{Fr}_{i-1}) \multimap \exists \Gamma_i.(\Psi_i; \overrightarrow{Fr}_i; \mathcal{Rep}_{C_i}\langle H_i \rangle)\ \forall i \in 1..n}{\begin{array}{l} \Gamma \vdash I_1\ \ldots\ I_n\ \mathbf{newhist}\ C, k : (\Psi_0; \overrightarrow{Fr}_0) \multimap \exists \Gamma_1, \ldots, \Gamma_n. \\ \qquad (\Psi_n; \overrightarrow{Fr}_n; \mathit{HC}_{C,k}(\mathcal{Rep}_{C_1}\langle H_1 \rangle, \ldots, \mathcal{Rep}_{C_n}\langle H_n \rangle)) \end{array}} \tag{51}$$

$$\frac{\begin{array}{c} \Gamma_1, \Gamma' \vdash I : (\Psi_1; \overrightarrow{Fr}_1) \multimap \exists \Gamma_2.(\Psi_2; \overrightarrow{Fr}_2; \tau) \\ \Psi_1' \preceq \Psi_1 \quad \overrightarrow{Fr}_1' \preceq \overrightarrow{Fr}_1 \quad \Psi_2 \preceq \Psi_2' \quad \overrightarrow{Fr}_2 \preceq \overrightarrow{Fr}_2' \quad \tau \preceq \tau' \end{array}}{\Gamma_1, \Gamma' \vdash I : (\Psi_1'; \overrightarrow{Fr}_1') \multimap \exists \Gamma_2, \Gamma'.(\Psi_2'; \overrightarrow{Fr}_2'; \tau')} \tag{52}$$

$$\frac{}{\Gamma \vdash \boxed{\mathbf{0}} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; \mathbf{void})} \tag{53}$$

$$\frac{}{\Gamma \vdash \boxed{i4} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; \mathbf{int32})} \tag{54}$$

$$\frac{\Psi = \Psi' \star (\ell \mapsto H)}{\Gamma, \ell{:}C \vdash \boxed{\ell} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; C\langle \ell \rangle)} \tag{55}$$

$$\frac{}{\Gamma, \ell{:}C\langle ? \rangle \vdash \boxed{\ell} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; C\langle ? \rangle)} \tag{56}$$

$$\frac{}{\Gamma \vdash \boxed{\mathit{rep}_C(H)} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; \mathcal{Rep}_C\langle H \rangle)} \tag{57}$$

$$\frac{\Gamma \vdash I : (\Psi; \overrightarrow{Fr}) \multimap \exists \Gamma'.(\Psi'; \overrightarrow{Fr}'\,\mathit{Fr}_0; \tau)}{\Gamma \vdash I\ \mathbf{ret} : (\Psi; \overrightarrow{Fr}) \multimap \exists \Gamma'.(\Psi'; \overrightarrow{Fr}'; \tau)} \tag{58}$$

The judgment $\Gamma \vdash \mathit{Sig}_1 <: \mathit{Sig}_2$ in rule 45 asserts that Sig_1 alpha-varies to Sig_2. That is, there exists a substitution $\sigma : \ell \to \ell$ such that $\sigma(\mathit{Sig}_1) = \mathit{Sig}_2$ and any free variables in Sig_2 are drawn from Γ. This captures the requirement that call sites must satisfy the callee's precondition and can assume the callee's postcondition.

B Proofs

The proofs of Terminating Policy Adherence (Theorem 1) and of Non-terminating Prefix Adherence (Theorem 2) are arrived at in three steps.

First, in §B.2 we prove subject reduction for the type system. That is, we prove that taking a step according to the operational semantics provided in Figure 7 preserves the type of a MOBILE term as defined in Appendix A. Second, in §B.3 we prove that well-typed MOBILE terms can take a step as long as they have not been reduced to a value or have not entered a "bad" state, such as by performing an **unpack** operation on an empty package. Third, these two results are leveraged in §B.4 to prove Terminating Policy Adherence and Non-terminating Prefix Adherence theorems. That is, we show that well-typed MOBILE programs that terminate normally will satisfy the security policy, and that well-typed MOBILE programs that do not terminate or that enter a "bad" state will satisfy a prefix of the security policy.

B.1 Canonical Derivations

In the proofs that follow, it will be useful to appeal to the following "obvious" facts about the derivation system given in Figure 9. (Proofs of the facts below can be obtained by trivial inductions over the derivations of the various relevant judgments.)

Fact 1. If $\Gamma' \vdash_{heap} h : \Gamma$ holds then the following three statements are equivalent:

(i) $\Gamma = \Gamma_0, \ell{:}C$

(ii) $h = h_0, (\ell \mapsto obj_C\{f_i = v_i | i \in 1..\mathit{fields}(C)\}^{\vec{e}})$

(iii) There exists a derivation of $\Gamma \vdash_{heap} h : \Gamma$ that ends in

$$\frac{\begin{array}{c}\Gamma' \vdash_{heap} h_0 : \Gamma_0 \\ \Gamma' \vdash \boxed{v_i} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; \mathit{field}(C, f_i)) \ \forall i \in 1..\mathit{fields}(C)\end{array}}{\Gamma' \vdash_{heap} h : \Gamma}(24)$$

and the following three statements are equivalent:

(i) $\Gamma = \Gamma_0, \ell{:}C\langle ?\rangle$

(ii) $h = h_0, (\ell \mapsto pkg(\ldots))$

(iii) There exists a derivation of $\Gamma \vdash_{heap} h : \Gamma$ that ends in

$$\frac{\Gamma' \vdash_{heap} h_0 : \Gamma_0}{\Gamma' \vdash_{heap} h : \Gamma}(25)$$

Fact 2. If $\vdash_{hist} h : (\Gamma; \Psi)$ holds then the following three statements are equivalent:

(i) $\Gamma = \Gamma_0, \ell'{:}C$

(ii) $h = h_0, (\ell' \mapsto obj_C\{\ldots\}^{\vec{e}})$

(iii) There exists a derivation of $\vdash_{\mathit{hist}} h : (\Gamma; \Psi)$ that ends in one of

$$\frac{\vdash_{\mathit{hist}} h_0 : (\Gamma_0; \Psi_0) \qquad \overrightarrow{e} \subseteq H}{\vdash_{\mathit{hist}} h : (\Gamma; \Psi)}(28)\,,$$

$$\frac{\vdash_{\mathit{hist}} h_1 : (\Gamma_1; \Psi) \qquad \overrightarrow{e} \subseteq H \subseteq \mathit{policy}(C)}{\vdash_{\mathit{hist}} h : (\Gamma; \Psi)}(29)\,,\ \text{or}$$

$$\frac{\vdash_{\mathit{hist}} h_0 : (\Gamma_0; \Psi) \qquad \overrightarrow{e} \subseteq \mathit{policy}(C)}{\vdash_{\mathit{hist}} h : (\Gamma; \Psi)}(30)$$

where $\Psi = \Psi_0 \star (\ell' \mapsto H)$, $\Gamma_1 = \Gamma_0, \ell{:}C\langle ?\rangle$, and $h_1 = h_0, (\ell \mapsto \mathit{pkg}(\ell', \mathit{rep}_C(H)))$;

and the following three statements are equivalent:

(i) $\Gamma = \Gamma_0, \ell{:}C\langle ?\rangle$

(ii) $h = h_0, (\ell \mapsto \mathit{pkg}(\ldots))$

(iii) There exists a derivation of $\vdash_{\mathit{hist}} h : (\Gamma; \Psi)$ that ends in one of

$$\frac{\vdash_{\mathit{hist}} h_1 : (\Gamma_1; \Psi) \qquad \overrightarrow{e} \subseteq H \subseteq \mathit{policy}(C)}{\vdash_{\mathit{hist}} h : (\Gamma; \Psi)}(29)$$

or

$$\frac{\vdash_{\mathit{hist}} h_0 : (\Gamma_0; \Psi)}{\vdash_{\mathit{hist}} h : (\Gamma; \Psi)}(31)$$

where $\Gamma_1 = \Gamma_0, \ell{:}C\langle ?\rangle$ and $h_1 = h_0, (\ell \mapsto \mathit{pkg}(\ell', \mathit{rep}_C(H)))$.

Fact 3. The following judgments can be weakened in the following ways:

1. If $\Gamma_0 \vdash_{\mathit{heap}} h : \Gamma$ holds then $\Gamma_0, \Gamma' \vdash_{\mathit{heap}} h : \Gamma$ also holds.
2. If $\Gamma_0 \vdash_{\mathit{stack}} s : \overrightarrow{\mathit{Fr}}$ holds then $\Gamma_0, \Gamma' \vdash_{\mathit{stack}} s : \overrightarrow{\mathit{Fr}}$ also holds.
3. If $\Gamma_0 \vdash I : (\Psi; \overrightarrow{\mathit{Fr}}) \multimap \exists\Gamma''.(\Psi''; \overrightarrow{\mathit{Fr}}''; \tau)$ holds then $\Gamma_0, \Gamma' \vdash I : (\Psi; \overrightarrow{\mathit{Fr}}) \multimap \exists\Gamma''.(\Psi''; \overrightarrow{\mathit{Fr}}''; \tau)$ also holds.

Facts 1 and 2 state that when $\Gamma' \vdash_{\mathit{heap}} h : \Gamma$ holds or $\vdash_{\mathit{hist}} h : (\Gamma; \Psi)$ holds, then Γ and h match element for element, and there is a way to reorganize the derivation of either judgment to bring the rule that refers to any particular element to the bottom of the derivation tree. That is, the rule applications in either derivation can be reordered arbitrarily. Fact 3 states that judgment $\Gamma_0 \vdash_{\mathit{heap}} h : \Gamma$, judgment $\Gamma_0 \vdash_{\mathit{stack}} s : \overrightarrow{\mathit{Fr}}$, and judgment $\Gamma_0 \vdash I : (\Psi; \overrightarrow{\mathit{Fr}}) \multimap \exists\Gamma''.(\Psi''; \overrightarrow{\mathit{Fr}}''; \tau)$ can be weakened by adding more elements to Γ_0.

B.2 Subject Reduction

Lemma 1 (Context Widening). *If* $\Gamma \vdash I : (\Psi; \mathit{Fr}) \multimap \exists\Gamma'.(\Psi'; \mathit{Fr}'; \tau)$ *holds and* I *contains no* **ret** *instructions, then* $\Gamma \vdash I : (\Psi_{\mathit{extra}} \star \Psi; \overrightarrow{\mathit{Fr}}\,\mathit{Fr}) \multimap \exists\Gamma'.(\Psi_{\mathit{extra}} \star \Psi'; \overrightarrow{\mathit{Fr}}\,\mathit{Fr}'; \tau)$ *holds.*

Proof. Observe that all typing rules except the typing rule for **ret** (58) are parameterized by an arbitrary frame list prefix that remains unchanged by an application of the rule. Since I has no **ret** instructions, this suffices to prove that $\Gamma \vdash I : (\Psi; \overrightarrow{\mathit{Fr}}\,\mathit{Fr}) \multimap \exists\Gamma'.(\Psi'; \overrightarrow{\mathit{Fr}}\,\mathit{Fr}'; \tau)$ holds.

It remains to show that $\Psi \vdash I : (\Psi_{\mathit{extra}} \star \Psi; \mathit{Fr}) \multimap \exists\Gamma'.(\Psi_{\mathit{extra}} \star \Psi'; \mathit{Fr}'; \tau)$ holds. Let $\mathcal{D}$ be the derivation of $\Gamma \vdash I : (\Psi; \mathit{Fr}) \multimap \exists\Gamma'.(\Psi'; \mathit{Fr}'; \tau)$. Proof is by induction on the structure of $\mathcal{D}$.

Case 1: $\mathcal{D}$ ends in rule 36, 42, 47, 53, 54, 55, 56, or 57. In these cases, $\Psi' = \Psi$. The lemma follows immediately by instantiating Ψ with $\Psi_{\mathit{extra}} \star \Psi$ in each typing rule.

Case 2: $\mathcal{D}$ ends in rule 37, 38, 39, 40, 41, 43, 44, 46, 48, 49, 50, or 51. The lemma follows by inductive hypothesis, by instantiating each antecedent of the form $\Gamma_0 \vdash I_0 : (\Psi_0; \mathit{Fr}_0) \multimap \exists\Gamma'_0.(\Psi'_0; \mathit{Fr}'_0; \tau_0)$ with $\Gamma_0 \vdash I_0 : (\Psi_{\mathit{extra}} \star \Psi_0; \mathit{Fr}_0) \multimap \exists\Gamma'_0.(\Psi_{\mathit{extra}} \star \Psi'_0; \mathit{Fr}'_0; \tau_0)$.

Case 3: $\mathcal{D}$ ends in rule 45. In addition to instantiating into each antecedent as in the previous case, instantiate Ψ_{unused} with $\Psi_{\mathit{extra}} \star \Psi_{\mathit{unused}}$. The lemma then holds by inductive hypothesis.

Case 4: $\mathcal{D}$ ends in rule 52. Observe from the subtyping rules that if $\Psi_1 \preceq \Psi'_1$ then $\Psi_{\mathit{extra}} \star \Psi_1 \preceq \Psi_{\mathit{extra}} \star \Psi'_1$. We can therefore instantiate the rule's antecedents as in the previous two cases to prove the lemma by inductive hypothesis.

□

Lemma 2 (Context Subtyping). *If* $\vdash_{\mathit{hist}} h : (\Gamma; \Psi)$ *and* $\Psi \preceq \Psi'$ *hold then* $\vdash_{\mathit{hist}} h : (\Gamma; \Psi')$ *holds.*

Proof. Let $\mathcal{D}$ be a derivation of $\vdash_{\mathit{hist}} h : (\Gamma; \Psi)$. Proof is by induction over the structure of $\mathcal{D}$.

Base Case: If $\mathcal{D}$ ends with rule 33, then $\Psi = \Psi' = \cdot$ and the lemma holds immediately.

Inductive Case: If $\mathcal{D}$ ends in any remaining rule other than rule 28, then the lemma follows immediately from the inductive hypothesis. Assume $\mathcal{D}$ ends in rule 28 and therefore has the form

$$\frac{\vdash_{\mathit{hist}} h_0 : (\Gamma_0; \Psi_0) \qquad \overrightarrow{e} \subseteq H}{\vdash_{\mathit{hist}} h : (\Gamma; \Psi)}(28)$$

where $\Gamma = \Gamma_0, \ell{:}C$, $h = h_0, (\ell \mapsto obj_C\{\ldots\}^{\overrightarrow{e}})$, and $\Psi = \Psi_0 \star (\ell \mapsto H)$. Since $\Psi \preceq \Psi'$, it follows that $\Psi' = \Psi_0' \star (\ell \mapsto H')$ such that $H \subseteq H'$ and $\Psi_0 \preceq \Psi_0'$. Thus, by inductive hypothesis one can derive

$$\frac{\vdash_{hist} h_0 : (\Gamma_0; \Psi_0') \qquad \overrightarrow{e} \subseteq H'}{\vdash_{hist} h : (\Gamma; \Psi')} (28)$$

□

Lemma 3 (Stepwise Subject Reduction)**.** *Assume that*

$$\Gamma \vdash \psi : (\Psi; \overrightarrow{Fr}) \tag{59}$$

$$\Gamma \vdash I : (\Psi; \overrightarrow{Fr}) \multimap \exists\Gamma''.(\Psi''; \overrightarrow{Fr}''; \tau) \tag{60}$$

both hold and assume that all methods in $Dom(methodbody)$, are well-typed. If $\psi, I \rightsquigarrow \psi', I'$ holds then there exists Γ', Ψ', $\overrightarrow{Fr}'$, and $\sigma : \theta \to \overrightarrow{e}$ such that $\Gamma' \vdash \psi' : (\Psi'; \overrightarrow{Fr}')$ holds and $\Gamma' \vdash I' : (\Psi'; \overrightarrow{Fr}') \multimap \exists\Gamma''.(\sigma(\Psi''); \sigma(\overrightarrow{Fr}''); \sigma(\tau))$ holds.

Proof. Proof is by induction on the derivation of the judgment $\psi, I \rightsquigarrow \psi', I'$. To make the proof more tractable, in what follows we make the simplifying assumption that weakening rule 52 does not appear in the derivation of judgment 60. Similar logic to that presented below applies to cases where rule 52 is present.

Case 1: $\psi, \textbf{ldc.i4}\ i4 \rightsquigarrow \psi, \boxed{i4}$. Then $\Gamma'' = \cdot$ and $(\Psi''; \overrightarrow{Fr}''; \tau) = (\Psi; \overrightarrow{Fr}; \textbf{int32})$ by 36. To satisfy the lemma, choose $\Gamma' = \Gamma$, $\Psi' = \Psi$, $\overrightarrow{Fr}' = \overrightarrow{Fr}$, and $\sigma = \cdot$ and apply typing rule 54.

Case 2: $\psi, E[I_0] \rightsquigarrow \psi', E[I_0']$. Let $\mathcal{D}$ be a derivation of 60. Observe that for all possible $E[I_0]$, derivation $\mathcal{D}$ includes a subderivation $\mathcal{D}_2$ of $\Gamma \vdash I_0 : (\Psi; \overrightarrow{Fr}) \multimap \exists\Gamma_2'.(\Psi_2'; \overrightarrow{Fr}_2'; \tau_2)$. By inductive hypothesis, there exists Γ', Ψ', $\overrightarrow{Fr}'$, and σ such that $\Gamma' \vdash \psi' : (\Psi'; \overrightarrow{Fr}')$ and $\Gamma' \vdash I_0' : (\Psi'; \overrightarrow{Fr}') \multimap \exists\Gamma_2'.(\sigma(\Psi_2'); \sigma(\overrightarrow{Fr}_2'); \sigma(\tau_2))$. Let $\mathcal{D}_2'$ be a derivation of this latter judgment. Then derivation $\mathcal{D}$ can be modified by replacing subderivation $\mathcal{D}_2$ with derivation $\mathcal{D}_2'$ to obtain a derivation of $\Gamma' \vdash E[I_0'] : (\Psi'; \overrightarrow{Fr}') \multimap \exists\Gamma''.(\sigma(\Psi''); \sigma(\overrightarrow{Fr}''); \sigma(\tau))$.

Case 3: $\psi, \boxed{i4}\ I_2\ I_3\ \textbf{cond} \rightsquigarrow \psi, I_j$ where $j \in \{2, 3\}$. Any derivation of 60 contains a subderivation of $\Gamma \vdash I_j : (\Psi; \overrightarrow{Fr}) \multimap \exists\Gamma''.(\Psi''; \overrightarrow{Fr}''; \tau)$ (by 37 and 54). Thus the lemma is satisfied by choosing $\Gamma' = \Gamma$, $\Psi' = \Psi$, $\overrightarrow{Fr}' = \overrightarrow{Fr}$, and $\sigma = \cdot$.

Case 4: $\psi, I_1\ I_2\ \mathbf{while} \rightsquigarrow \psi, I_1\ (I_2; (I_1\ I_2\ \mathbf{while}))\ \boxed{\mathbf{0}}\ \mathbf{cond}$. Any derivation of 60 must have the form

$$\dfrac{\dfrac{\Gamma \vdash I_1 : (\Psi; \overrightarrow{Fr}) \multimap \exists\Gamma_1.(\Psi_1; \overrightarrow{Fr}_1; \mathbf{int32}) \qquad \Gamma \vdash I_2 : (\Psi_1; \overrightarrow{Fr}_1) \multimap \exists\Gamma_2.(\Psi; \overrightarrow{Fr}; \mathbf{void}) \qquad \Gamma \vdash \boxed{\mathbf{0}} : (\Psi_1; \overrightarrow{Fr}_1) \multimap \exists\Gamma_2.(\Psi; \overrightarrow{Fr}; \mathbf{void})}{\Gamma \vdash I_1\ I_2\ \boxed{\mathbf{0}}\ \mathbf{cond} : (\Psi; \overrightarrow{Fr}) \multimap \exists\Gamma''.(\Psi; \overrightarrow{Fr}; \mathbf{void})}(37)}{\Gamma \vdash I_1\ I_2\ \mathbf{while} : (\Psi; \overrightarrow{Fr}) \multimap \exists\Gamma''.(\Psi; \overrightarrow{Fr}; \mathbf{void})}(38)$$

where $\Gamma = \Gamma_0, \Gamma''$ and $\Gamma'' = \Gamma_1, \Gamma_2$. One can therefore derive

$$\dfrac{\dfrac{\Gamma \vdash I_1 : (\Psi; \overrightarrow{Fr}) \multimap \exists\Gamma_1.(\Psi_1; \overrightarrow{Fr}_1; \mathbf{int32})}{\Gamma \vdash I_1 : (\Psi; \overrightarrow{Fr}) \multimap \exists\Gamma''.(\Psi_1; \overrightarrow{Fr}_1; \mathbf{int32})}(52) \qquad \dfrac{\dfrac{\Gamma \vdash I_2 : (\Psi_1; \overrightarrow{Fr}_1) \multimap \exists\Gamma_2.(\Psi; \overrightarrow{Fr}; \mathbf{void})}{\Gamma \vdash I_2 : (\Psi_1; \overrightarrow{Fr}_1) \multimap \exists\Gamma''.(\Psi; \overrightarrow{Fr}; \mathbf{void})}(52) \qquad \Gamma \vdash I_1\ I_2\ \mathbf{while} : (\Psi; \overrightarrow{Fr}) \multimap \exists\Gamma''.(\Psi; \overrightarrow{Fr}; \mathbf{void})}{\Gamma \vdash I_2; (I_1\ I_2\ \mathbf{while}) : (\Psi_1; \overrightarrow{Fr}_1) \multimap \exists\Gamma''.(\Psi; \overrightarrow{Fr}; \mathbf{void})}(38) \qquad \dfrac{\Gamma \vdash \boxed{\mathbf{0}} : (\Psi_1; \overrightarrow{Fr}_1) \multimap \exists\Gamma_2.(\Psi; \overrightarrow{Fr}; \mathbf{void})}{\Gamma \vdash \boxed{\mathbf{0}} : (\Psi_1; \overrightarrow{Fr}_1) \multimap \exists\Gamma''.(\Psi; \overrightarrow{Fr}; \mathbf{void})}(52)}{\Gamma_0, \Gamma'' \vdash I_1\ (I_2; (I_1\ I_2\ \mathbf{while}))\ \boxed{\mathbf{0}}\ \mathbf{cond} : (\Psi; \overrightarrow{Fr}) \multimap \exists\Gamma''.(\Psi; \overrightarrow{Fr}; \mathbf{void})}(37)$$

The lemma is thus satisfied by choosing $\Gamma' = \Gamma$, $\Psi' = \Psi$, $\overrightarrow{Fr}' = \overrightarrow{Fr}$, and $\sigma = \cdot$.

Case 5: $\psi, \boxed{v}; I_2 \rightsquigarrow \psi, I_2$. Any derivation of 60 contains a subderivation of $\Gamma \vdash I_2 : (\Psi; \overrightarrow{Fr}) \multimap \exists\Gamma''.(\Psi''; \overrightarrow{Fr}''; \tau)$ (by 39 and 53). Thus the lemma is satisfied by choosing $\Gamma' = \Gamma$, $\Psi' = \Psi$, $\overrightarrow{Fr}' = \overrightarrow{Fr}$, and $\sigma = \cdot$.

Case 6: $\psi, \mathbf{ldarg}\ j \rightsquigarrow \psi, \boxed{v_j}$ where $\psi = (h, s(v_0, \ldots, v_n))$. From 60 and 42, $\overrightarrow{Fr}$ has the form $\overrightarrow{Fr}_0 Fr$ and $0 \leq j \leq n$. From 59 and 34, $Fr = (\tau_0, \ldots, \tau_n)$ and $\Gamma \vdash \boxed{v_j} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; \tau_j)$ holds. The lemma is therefore satisfied by choosing $\Gamma' = \Gamma$, $\Psi' = \Psi$, $\overrightarrow{Fr}' = \overrightarrow{Fr}$, and $\sigma = \cdot$.

Case 7: $(h, s), \boxed{v}\ \mathbf{starg}\ j \rightsquigarrow (h, s'), \boxed{\mathbf{0}}$ where $s = s_0(v_0, \ldots, v_n)$ for some stack prefix s_0, and $s' = s_0(v_0, \ldots, v_{j-1}, v, v_{j+1}, \ldots, v_n)$. From 60 and 43, $\overrightarrow{Fr}$ has the form $\overrightarrow{Fr}_0 Fr$, and $0 \leq j \leq n$. From 59 and 34, $Fr = (\tau_0, \ldots, \tau_n)$. From 60 and 43, $\Gamma'' = \cdot$, $\Psi'' = \Psi$, $\overrightarrow{Fr}'' = \overrightarrow{Fr}_0(\tau_0, \ldots, \tau_{j-1}, \tau', \tau_{j+1}, \ldots, \tau_n)$, $\tau = \mathbf{void}$, and $\Gamma \vdash \boxed{v} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; \tau')$ holds. Choose $\Gamma' = \Gamma$, $\Psi' = \Psi$, $\overrightarrow{Fr}' = \overrightarrow{Fr}''$, and $\sigma = \cdot$. Since all type judgments for value expressions are independent of frames, one can derive $\Gamma \vdash \boxed{v} : (\Psi'; \overrightarrow{Fr}') \multimap (\Psi'; \overrightarrow{Fr}'; \tau')$ to prove by 34 that $\Gamma' \vdash (h; s') : (\Psi'; \overrightarrow{Fr}')$ holds. Furthermore, $\Gamma' \vdash \boxed{\mathbf{0}} : (\Psi'; \overrightarrow{Fr}') \multimap (\Psi''; \overrightarrow{Fr}''; \tau)$ holds by 53, satisfying the lemma.

Case 8: $(h, s), \boxed{v_1}\ \ldots\ \boxed{v_n}\ \mathbf{newobj}\ C(\mu_1, \ldots, \mu_n) \rightsquigarrow (h', s), \boxed{\ell}$ where $h' = h, (\ell \mapsto obj_C\{f_i = v_i | i \in 1..n\}^{\epsilon})$ and $n = \mathit{fields}(C)$. From 60 and 44, $\Gamma'' = \ell{:}C$, $\Psi'' = \Psi \star (\ell \mapsto \epsilon)$, $\overrightarrow{Fr}'' = \overrightarrow{Fr}$, and $\tau = C\langle\ell\rangle$. Additionally, $\Gamma \vdash \boxed{v_i} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; \mathit{field}(C, f_i))\ \forall i \in 1..n$. Choose $\Gamma' = \Gamma''$, $\Psi' = \Psi''$, $\overrightarrow{Fr}' = \overrightarrow{Fr}''$, and $\sigma = \cdot$. From 59 one can derive

$$\dfrac{\Gamma' \vdash_{\mathit{heap}} h : \Gamma \qquad \Gamma' \vdash \boxed{v_i} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; \mathit{field}(C, f_i))\ \forall i \in 1..n}{\Gamma' \vdash_{\mathit{heap}} h' : \Gamma'}(24)$$

and derive

$$\frac{\vdash_{hist} h : (\Gamma'; \Psi) \qquad \epsilon \subseteq \epsilon}{\vdash_{hist} h' : (\Gamma'; \Psi')}(24)$$

Thus $\Gamma' \vdash (h', s) : (\Psi', \overrightarrow{Fr}')$ holds. Further, observe that $\Gamma' \vdash \boxed{\ell} : (\Psi'; \overrightarrow{Fr}') \multimap \exists\Gamma''.(\Psi''; Fr''; C\langle\ell\rangle)$ holds by 55 because $\Gamma' = \Gamma, \ell{:}C$. Thus the lemma is satisfied.

Case 9: $(h, s), \boxed{v_0}\ \ldots\ \boxed{v_n}$ **callvirt** $C{::}m.Sig \rightsquigarrow (h, sa), I_0$ **ret** where $a = (v_0, \ldots, v_n)$ and $I_0 = \mathit{methodbody}(C{::}m.Sig)$. From 60 and 45, $\overrightarrow{Fr}'' = \overrightarrow{Fr}$, and there exists $(\Psi_{in}, (\tau_0, \ldots, \tau_n))$, Ψ_{out}, Ψ_{unused}, and Fr_{out} such that $\Psi = \Psi_{unused} \star \Psi_{in}$, $\Psi'' = \Psi_{unused} \star \Psi_{out}$,

$$\Gamma \vdash \boxed{v_i} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; \tau_i)\ \forall i \in 0..n \tag{61}$$

and

$$\Gamma \vdash \mathit{Sig} <: (\Psi_{in}; (\tau_0, \ldots, \tau_n)) \multimap \exists\Gamma''.(\Psi_{out}; Fr_{out}; \tau).$$

Since $C{::}m.Sig$ is well-typed, it follows that

$$\Gamma \vdash I_0 : (\Psi_{in}; (\tau_0, \ldots, \tau_n)) \multimap \exists\Gamma''.(\Psi_{out}; Fr_{out}; \tau).$$

By context widening, this implies that

$$\Gamma \vdash I_0 : (\Psi_{unused} \star \Psi_{in}; \overrightarrow{Fr}(\tau_0, \ldots, \tau_n)) \multimap \\ \exists\Gamma''.(\Psi_{unused} \star \Psi_{out}; \overrightarrow{Fr}\, Fr_{out}; \tau)$$

which collapses to

$$\Gamma \vdash I_0 : (\Psi; \overrightarrow{Fr}(\tau_0, \ldots, \tau_n)) \multimap \exists\Gamma''.(\Psi''; \overrightarrow{Fr}\, Fr_{out}; \tau).$$

Choose $\Gamma' = \Gamma$, $\Psi' = \Psi$, $\overrightarrow{Fr}' = \overrightarrow{Fr}(\tau_0, \ldots, \tau_n)$, and $\sigma = \cdot$. To prove that $\Gamma' \vdash I' : (\Psi'; \overrightarrow{Fr}') \multimap \exists\Gamma''.(\Psi''; \overrightarrow{Fr}''; \tau)$ holds, derive

$$\frac{\Gamma \vdash I_0 : (\Psi; \overrightarrow{Fr}(\tau_0, \ldots, \tau_n)) \multimap \exists\Gamma''.(\Psi''; \overrightarrow{Fr}\, Fr_{out}; \tau)}{\Gamma \vdash I_0\ \textbf{ret} : (\Psi; \overrightarrow{Fr}(\tau_0, \ldots, \tau_n)) \multimap \exists\Gamma''.(\Psi''; \overrightarrow{Fr}; \tau)}(58)$$

(recalling that $\overrightarrow{Fr}'' = \overrightarrow{Fr}$). To prove that $\Gamma' \vdash (h; sa) : (\Psi'; \overrightarrow{Fr}')$ holds, observe that $\Gamma \vdash_{stack} s : \overrightarrow{Fr}$ holds by 59, and therefore one can derive

$$\frac{\Gamma \vdash_{stack} s : \overrightarrow{Fr} \qquad \Gamma \vdash \boxed{v_i} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; \tau_i)\ \forall i \in 0..n}{\Gamma' \vdash_{stack} sa : \overrightarrow{Fr}'}(34)$$

by 61 and 34.

Case 10: $(h, sa), \boxed{v}$ **ret** $\leadsto (h, s), \boxed{v}$. By 60 and 58, $\Gamma'' = \cdot$, $\Psi = \Psi''$, and $\overrightarrow{Fr} = \overrightarrow{Fr}'' Fr_0$ for some Fr_0. Choose $\Gamma' = \Gamma$, $\Psi' = \Psi''$, $\overrightarrow{Fr}' = \overrightarrow{Fr}''$, and $\sigma = \cdot$. Any derivation of 60 has a subderivation of $\Gamma \vdash \boxed{v} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; \tau)$. Since the typing rules for value expressions are independant of frame, one can therefore derive $\Gamma \vdash \boxed{v} : (\Psi'; \overrightarrow{Fr}') \multimap (\Psi''; \overrightarrow{Fr}''; \tau)$. Furthermore, one derivation of 59 has a subderivation of

$$\frac{\Gamma \vdash_{stack} s : \overrightarrow{Fr}'' \qquad \vdots}{\Gamma \vdash_{stack} sa : \overrightarrow{Fr}'' Fr_0}(34)$$

Hence $\Gamma' \vdash_{stack} s : \overrightarrow{Fr}'$ holds.

Case 11: $(h, s), \boxed{\ell}$ **ldfld** $\mu\ C{::}f \leadsto (h, s), \boxed{v}$ where $h(\ell) = obj_C\{\ldots, f = v, \ldots\}^{\overrightarrow{e}}$. By 60 and 40, $\Gamma'' = \cdot$, $\Psi = \Psi''$, and $\overrightarrow{Fr} = \overrightarrow{Fr}''$. Choose $\Gamma' = \Gamma$, $\Psi' = \Psi$, $\overrightarrow{Fr}' = \overrightarrow{Fr}$, and $\sigma = \cdot$. Any derivation of 60 has a subderivation of $\Gamma \vdash \boxed{v} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; \tau)$. Hence $\Gamma \vdash \boxed{v} : (\Psi'; \overrightarrow{Fr}') \multimap (\Psi''; \overrightarrow{Fr}''; \tau)$ holds. Furthermore, $\Gamma' \vdash (h; s) : (\Psi'; \overrightarrow{Fr}')$ holds by 59.

Case 12: $(h, s), \boxed{\ell}\ \boxed{v}$ **stfld** $\mu\ C{::}f_j \leadsto (h', s), \boxed{\mathbf{0}}$ where $h' = h[\ell \mapsto obj_C[f_j \mapsto v]]$, and $1 \le j \le \mathit{fields}(C)$, and $h(\ell) = obj_C\{f_i = v_i | i \in 1..\mathit{fields}(C)\}^{\overrightarrow{e}}$. By 60 and 41, $\Gamma'' = \cdot$, $\Psi = \Psi''$, $\overrightarrow{Fr} = \overrightarrow{Fr}''$, and $\tau =$ **void**. Choose $\Gamma' = \Gamma$, $\Psi' = \Psi$, $\overrightarrow{Fr}' = \overrightarrow{Fr}$, and $\sigma = \cdot$. Observe that $\Gamma' \vdash \boxed{\mathbf{0}} : (\Psi'; \overrightarrow{Fr}') \multimap (\Psi''; \overrightarrow{Fr}''; \tau)$ holds by rule 53. Furthermore, since one derivation of 59 has a subderivation of

$$\frac{\begin{array}{c}\Gamma \vdash_{heap} h_0 : \Gamma_0 \\ \Gamma \vdash \boxed{v_i} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; \mathit{field}(C, f_i))\ \forall i \in 1..\mathit{fields}(C)\end{array}}{\Gamma \vdash_{heap} h : \Gamma}(24)$$

where $\Gamma = \Gamma_0, \ell{:}C$ and $h = h_0, (\ell \mapsto obj_C\{\ldots\}^{\overrightarrow{e}})$, and since 60 implies that all three of $\Gamma \vdash \boxed{v} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; \mu)$, $\mathit{field}(C, f_j) = \mu$, and $\ell \in Dom(\Gamma')$ hold, one can derive

$$\frac{\begin{array}{c}\Gamma \vdash_{heap} h_0 : \Gamma_0 \\ \Gamma \vdash \boxed{v} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; \mathit{field}(C, f_j)) \\ \Gamma \vdash \boxed{v_i} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; \mathit{field}(C, f_i))\ \forall i \in 1..j-1, j+1..\mathit{fields}(C)\end{array}}{\Gamma' \vdash_{heap} h' : \Gamma'}(24)$$

Hence $\Gamma' \vdash (h'; s) : (\Psi'; \overrightarrow{Fr}')$ holds.

Case 13: $(h, s), \boxed{\ell}$ **evt** $e_1 \leadsto (h', s), \boxed{\mathbf{0}}$ where $h' = h[\ell \mapsto obj_C\{\ldots\}^{\overrightarrow{e}e_1}$ and $h(\ell) = obj_C\{\ldots\}^{\overrightarrow{e}}$. By 60 and 46, $\Gamma'' = \cdot$, $\overrightarrow{Fr} = \overrightarrow{Fr}''$, $\tau =$ **void**,

$\Psi = \Psi_1 \star (\ell \mapsto H)$ for some Ψ_1 and H, and $\Psi'' = \Psi_1 \star (\ell \mapsto He_1)$. Choose $\Gamma' = \Gamma$, $\Psi' = \Psi''$, $\overrightarrow{Fr}' = \overrightarrow{Fr}$, and $\sigma = \cdot$. Then $\Gamma' \vdash \boxed{\mathbf{0}} : (\Psi'; \overrightarrow{Fr}') \multimap (\Psi''; \overrightarrow{Fr}''; \tau)$ holds by typing rule 53. Furthermore, since one derivation of 59 has a subderivation of

$$\frac{\vdash_{\mathit{hist}} h_0 : (\Gamma_0; \Psi_1) \qquad \overrightarrow{e} \subseteq H}{\vdash_{\mathit{hist}} h : (\Gamma; \Psi)}(28)$$

where $\Gamma = \Gamma_0, \ell{:}C$ and $h = h_0, \ell \mapsto \mathit{obj}_C\{\ldots\}^{\overrightarrow{e}}$, one can derive

$$\frac{\vdash_{\mathit{hist}} h_0 : (\Gamma_0; \Psi_1) \qquad \overrightarrow{e}e_1 \subseteq He_1}{\vdash_{\mathit{hist}} h' : (\Gamma; \Psi'')}(28)$$

Hence $\Gamma' \vdash (h'; s) : (\Psi'; \overrightarrow{Fr}')$ holds.

Case 14: $(h, s), \mathbf{newpackage}\ C \rightsquigarrow (h', s), \boxed{\ell}$ where $h' = h, \ell \mapsto \mathit{pkg}(\cdot)$. By 60 and 47, $\Gamma'' = \ell{:}C\langle ?\rangle$, $\Psi = \Psi''$, $\overrightarrow{Fr} = \overrightarrow{Fr}''$, and $\tau = C\langle ?\rangle$. Choose $\Gamma' = \Gamma, \Gamma''$, $\Psi' = \Psi$, $\overrightarrow{Fr}' = \overrightarrow{Fr}$, and $\sigma = \cdot$. Observe that $\Gamma' \vdash \boxed{\ell} : (\Psi''; \overrightarrow{Fr}'') \multimap (\Psi''; \overrightarrow{Fr}''; \tau)$ by typing rule 56. Hence $\Gamma' \vdash \boxed{\ell} : (\Psi'; \overrightarrow{Fr}') \multimap \exists\Gamma''.(\Psi''; \overrightarrow{Fr}''; \tau)$ holds by rule 52. In addition, any derivation of 59 includes subderivations of $\Gamma \vdash_{\mathit{heap}} h : \Gamma$ and $\vdash_{\mathit{hist}} h : (\Gamma; \Psi)$; hence one can derive

$$\frac{\Gamma' \vdash_{\mathit{heap}} h : \Gamma}{\Gamma' \vdash_{\mathit{heap}} h' : \Gamma'}(25) \qquad \text{and} \qquad \frac{\vdash_{\mathit{hist}} h : (\Gamma; \Psi)}{\vdash_{\mathit{hist}} h' : (\Gamma'; \Psi')}(31)$$

Thus $\Gamma' \vdash (h'; s) : (\Psi'; \overrightarrow{Fr}')$ holds, proving the lemma.

Case 15: $\boxed{\ell}\ \boxed{\ell'}\ \boxed{\mathit{rep}_C(H)}\ \mathbf{pack} \rightsquigarrow (h', s), \boxed{\mathbf{0}}$ where $h(\ell) = \mathit{pkg}(\ldots)$ and $h' = h[\ell \mapsto \mathit{pkg}(\ell', \mathit{rep}_C(H))]$. By 60 and 48, $\Gamma'' = \cdot$, $\overrightarrow{Fr} = \overrightarrow{Fr}''$, $\tau = \mathbf{void}$, and $\Psi = \Psi'' \star (\ell \mapsto H')$ for some H'. Any derivation of 60 has subderivations of $\Gamma \vdash \boxed{\mathit{rep}_C(H)} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi''; \overrightarrow{Fr}; \mathcal{R}ep_C\langle H\rangle)$ (by rule 57) such that $H' \subseteq H \subseteq \mathit{policy}(C)$ (by rule 48), and of

$$\frac{\Psi = \Psi'' \star (\ell' \mapsto H')}{\Gamma \vdash \boxed{\ell'} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; C\langle \ell\rangle)}(55)$$

where $\Gamma = \Gamma_0, \ell{:}C\langle ?\rangle, \ell'{:}C$. One derivation of 59 has a subderivation of

$$\frac{\dfrac{\mathcal{D}}{\vdash_{\mathit{hist}} h_0, (\ell \mapsto \mathit{pkg}(\ldots)) : (\Gamma_0, \ell{:}C\langle ?\rangle; \Psi'')} \qquad \overrightarrow{e} \subseteq H'}{\vdash_{\mathit{hist}} h : (\Gamma; \Psi)}(28)$$

where $h = h_0, (\ell \mapsto \mathit{pkg}(\ldots)), (\ell' \mapsto \mathit{obj}_C\{\ldots\}^{\overrightarrow{e}})$ (because rule 28 is the only derivation rule that can add $\ell' \mapsto H'$ to Ψ.) Given the definition of h_0 above, observe that

$$h' = h_0, (\ell \mapsto \mathit{pkg}(\ell', \mathit{rep}_C(H))), (\ell' \mapsto \mathit{obj}_C\{\ldots\}^{\overrightarrow{e}})$$

and $\overrightarrow{e} \subseteq H' \subseteq H \subseteq \mathit{policy}(C)$. Choose $\Gamma' = \Gamma$, $\Psi' = \Psi''$, $\overrightarrow{Fr}' = \overrightarrow{Fr}$, and $\sigma = \cdot$. Observe that $\Gamma' \vdash \boxed{\mathbf{0}} : (\Psi'; \overrightarrow{Fr}') \multimap (\Psi''; \overrightarrow{Fr}''; \tau)$ is derivable using rule 53.

It remains to be shown that $\vdash_{\mathit{hist}} h' : (\Gamma'; \Psi')$ holds. To prove this, it suffices to prove that $\vdash_{\mathit{hist}} h_0 : (\Gamma_0; \Psi'')$ holds, since if this latter judgment holds, one can derive

$$\frac{\vdash_{\mathit{hist}} h_0 : (\Gamma_0; \Psi'') \qquad \overrightarrow{e} \subseteq H \subseteq \mathit{policy}(C)}{\vdash_{\mathit{hist}} h' : (\Gamma; \Psi'')}(29)$$

Suppose $h(\ell) = \mathit{pkg}(\cdot)$. Then

$$\mathcal{D} = \frac{\vdash_{\mathit{hist}} h_0 : (\Gamma_0; \Psi'')}{\vdash_{\mathit{hist}} h_0, (\ell \mapsto \mathit{pkg}(\cdot)) : (\Gamma_0, \ell{:}C\langle ? \rangle; \Psi'')}(31)$$

proving that $\vdash_{\mathit{hist}} h_0 : (\Gamma_0; \Psi'')$ holds.

Otherwise $h(\ell) = \mathit{pkg}(\ell'', \mathit{rep}_C(H''))$ for some ℓ'' and H''. In that case,

$$\mathcal{D} = \frac{\vdash_{\mathit{hist}} h_1 : (\Gamma_1; \Psi'') \qquad \overrightarrow{e''} \subseteq H'' \subseteq \mathit{policy}(C)}{\vdash_{\mathit{hist}} h_0, (\ell \mapsto \mathit{pkg}(\ell'', \mathit{rep}_C(H''))) : (\Gamma_0, \ell{:}C\langle ? \rangle; \Psi'')}(29)$$

where $\Gamma_0 = \Gamma_1, (\ell''{:}C)$ and $h_0 = h_1, (\ell'' \mapsto \mathit{obj}_C\{\ldots\}^{\overrightarrow{e''}})$. One can therefore derive

$$\frac{\vdash_{\mathit{hist}} h_1 : (\Gamma_1; \Psi'') \qquad \overrightarrow{e''} \subseteq \mathit{policy}(C)}{\vdash_{\mathit{hist}} h_0 : (\Gamma_0; \Psi'')}(30)$$

proving that $\vdash_{\mathit{hist}} h_0 : (\Gamma_0; \Psi'')$ holds.

Case 16: $(h, s(v_0, \ldots, v_n)), \boxed{\ell}$ **unpack** $j \rightsquigarrow (h[\ell \mapsto \mathit{pkg}(\cdot)], sa'), \boxed{\ell'}$ where $h(\ell) = \mathit{pkg}(\ell', \mathit{rep}_C(H))$ and $a' = (v_0, \ldots, v_{j-1}, \mathit{rep}_C(H), v_{j+1}, \ldots, v_n)$. By 60 and 51, $\Gamma'' = \ell{:}C, \theta$, $\Psi'' = \Psi \star (\ell \mapsto \theta)$, $\tau = C\langle \ell \rangle$, and $\overrightarrow{Fr}'' = \overrightarrow{Fr}_0(\tau_0, \ldots, \tau_{j-1}, \mathcal{Rep}_C\langle \theta \rangle, \tau_{j+1}, \ldots, \tau_n)$ where $\overrightarrow{Fr} = \overrightarrow{Fr}_0(\tau_0, \ldots, \tau_n)$. One derivation of 59 has a subderivation of

$$\frac{\vdash_{\mathit{hist}} h_0 : (\Gamma_0; \Psi) \qquad \overrightarrow{e} \subseteq H \subseteq \mathit{policy}(C)}{\vdash_{\mathit{hist}} h : (\Gamma; \Psi)}(29)$$

where $\Gamma = \Gamma_0, \ell{:}C\langle ?\rangle, \ell'{:}C$ and $h = h_0, (\ell \mapsto \mathit{pkg}(\ell', \mathit{rep}_C(H))), (\ell' \mapsto \mathit{obj}_C\{\ldots\}^{\overrightarrow{e}})$.

Choose $\Gamma' = \Gamma$, $\sigma = (\theta \mapsto \overrightarrow{e})$, $\Psi' = \sigma(\Psi'')$, and $\overrightarrow{Fr}' = \sigma(\overrightarrow{Fr}'')$. Since $\theta \notin \mathit{Dom}(\Gamma)$ (by rule 49), it follows that $\sigma(\Psi'') = \Psi \star (\ell \mapsto \overrightarrow{e})$. One can therefore derive

$$\frac{\Psi' = \Psi \star (\ell \mapsto \overrightarrow{e})}{\Gamma' \vdash \boxed{\ell'} : (\Psi'; \overrightarrow{Fr}') \multimap (\sigma(\Psi''); \sigma(\overrightarrow{Fr}''); \sigma(\tau))}(55)$$

and one can derive

$$\cfrac{\cfrac{\vdash_{\mathit{hist}} h_0 : (\Gamma_0; \Psi)}{\vdash_{\mathit{hist}} h_0, (\ell \mapsto \mathit{pkg}(\cdot)) : (\Gamma_0, \ell{:}C\langle ?\rangle; \Psi)}(31) \qquad \overrightarrow{e} \subseteq \overrightarrow{e}}{\vdash_{\mathit{hist}} h' : (\Gamma'; \Psi')}(28)$$

Finally, since any derivation of 59 has a subderivation of

$$\frac{\Gamma \vdash_{\mathit{stack}} s : \overrightarrow{Fr}_0 \qquad \Gamma \vdash \boxed{v_i} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; \tau_i)\ \forall i \in 0..n}{\Gamma \vdash_{\mathit{stack}} s(v_0, \ldots, v_n) : \overrightarrow{Fr}_0(\tau_0, \ldots, \tau_n)}(34)$$

and since $\Gamma \vdash \boxed{\mathit{rep}_C(H)} : (\Psi'; \overrightarrow{Fr}') \multimap (\Psi'; \overrightarrow{Fr}'; \mathcal{R}\mathit{ep}_C\langle H\rangle)$ holds by typing rule 57, it follows from derivation rule 34 that $\Gamma' \vdash_{\mathit{stack}} sa' : \overrightarrow{Fr}'$ holds.

Case 17: $\psi, \boxed{\mathit{rep}_C(H)}\ I_2\ I_3$ **condst** $C, k \rightsquigarrow \psi, I_j$ where $j \in \{2, 3\}$. Choose $\Gamma' = \Gamma$, $\overrightarrow{Fr}' = \overrightarrow{Fr}$,

$$\Psi' = \begin{cases} \mathit{ctx}^+_{C,k}(H, \Psi) & \text{if } j = 2 \\ \mathit{ctx}^-_{C,k}(H, \Psi) & \text{if } j = 3 \end{cases}$$

and $\sigma = \cdot$. By 60, 51, and 57, both $\Gamma \vdash I_2 : (\mathit{ctx}^+_{C,k}; \overrightarrow{Fr}) \multimap \exists\Gamma''.(\Psi''; \overrightarrow{Fr}''; \tau)$ and $\Gamma \vdash I_3 : (\mathit{ctx}^-_{C,k}; \overrightarrow{Fr}) \multimap \exists\Gamma''.(\Psi''; \overrightarrow{Fr}''; \tau)$ hold, so $\Gamma \vdash I_j : (\Psi'; \overrightarrow{Fr}') \multimap \exists\Gamma''.(\Psi''; \overrightarrow{Fr}''; \tau)$ holds.

Any derivation of 59 has subderivations of $\Gamma \vdash_{\mathit{heap}} h : \Gamma$ and $\vdash_{\mathit{hist}} h : (\Gamma; \Psi)$. To prove that $\Gamma' \vdash \psi : (\Psi'; \overrightarrow{Fr}')$, it suffices to show that $\vdash_{\mathit{hist}} h : (\Gamma; \Psi')$. If $j = 2$ then $\mathrm{test}_{C,k}(\mathit{rep}_C(H)) \neq 0$ (by 17), and axiom 20 therefore implies that $\Psi \preceq \mathit{ctx}^+_{C,k}(H, \Psi)$. Altnernatively, if $j = 3$ then $\mathrm{test}_{C,k}(\mathit{rep}_C(H)) = 0$, and axiom 19 therefore implies that $\Psi \preceq \mathit{ctx}^-_{C,k}(H, \Psi)$. In either case, $\Psi \preceq \Psi'$ holds. By context subtyping, we conclude that $\vdash_{\mathit{hist}} h : (\Gamma; \Psi')$ also holds.

Case 18: $\psi, \boxed{v_1}\ \ldots\ \boxed{v_n}$ **newhist** $C, k \rightsquigarrow \psi, \boxed{\mathit{hc}_{C,k}(v_1, \ldots, v_n)}$. By 60 and 51, $\Gamma'' = \cdot$, $\Psi = \Psi''$, $\overrightarrow{Fr} = \overrightarrow{Fr}''$, and $\tau = \mathit{HC}_{C,k}(\mathcal{R}\mathit{ep}_{C_1}\langle H_1\rangle, \ldots, \mathcal{R}\mathit{ep}_{C_n}\langle H_n\rangle)$

where $\Gamma \vdash \boxed{v_i} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; \mathcal{R}ep_{C_i}\langle H_i \rangle)$ holds for all $i \in 1..n$. Choose $\Gamma' = \Gamma$, $\Psi' = \Psi$, $\overrightarrow{Fr}' = \overrightarrow{Fr}$, and $\sigma = \cdot$. By axioms 21 and 22, there exists H such that $\tau = \mathcal{R}ep_C\langle H \rangle$ and $hc_{C,k}(v_1, \ldots, v_n) = rep_C(H)$. Thus, $\Gamma \vdash \boxed{rep_C(H)} : (\Psi'; \overrightarrow{Fr}') \multimap (\Psi''; \overrightarrow{Fr}''; \mathcal{R}ep_C\langle H \rangle)$ holds by typing rule 57.

□

Theorem 3 (Subject Reduction). *Assume that $\Gamma \vdash \psi : (\Psi; \overrightarrow{Fr})$ holds and assume that all methods in Dom(methodbody) are well-typed. If $\Gamma \vdash I : (\Psi; \overrightarrow{Fr}) \multimap \exists \Gamma''.(\Psi''; \overrightarrow{Fr}''; \tau)$ holds and $\psi, I \leadsto^n \psi', I'$ holds then there exist Γ', Ψ', $\overrightarrow{Fr}'$, and $\sigma : \theta \to \overrightarrow{e}$ such that $\Gamma' \vdash \psi' : (\Psi'; \overrightarrow{Fr}')$ holds and $\Gamma' \vdash I' : (\Psi'; Fr') \multimap (\sigma(\Psi''); \sigma(Fr''); \sigma(\tau))$ holds.*

Proof. Proof is by induction on n.

Base Case: Assume $n = 0$. Choose $\Gamma' = \Gamma$, $\Psi' = \Psi$, $\overrightarrow{Fr}' = \overrightarrow{Fr}$, and $\sigma = \cdot$. The theorem is then satisfied by assumption.

Inductive Case: Assume $n \geq 1$. Since $\psi, I \leadsto^n \psi', I'$ holds, there exist ψ_1 and I_1 such that $\psi, I \leadsto^{n-1} \psi_1, I_1$ holds and such that $\psi_1, I_1 \leadsto \psi', I'$ also holds. By inductive hypothesis, there exists Γ_1, Ψ_1, $\overrightarrow{Fr}_1$, and σ_1 such that $\Gamma_1 \vdash \psi_1 : (\Psi_1; \overrightarrow{Fr}_1)$ and $\Gamma_1 \vdash I_1 : (\Psi_1; \overrightarrow{Fr}_1) \multimap (\sigma_1(\Psi''); \sigma_1(\overrightarrow{Fr}''); \sigma_1(\tau))$ hold. The theorem then follows from the stepwise subject reduction lemma.

□

B.3 Progress

Theorem 4 (Progress). *Assume $\Gamma \vdash I : (\Psi; \overrightarrow{Fr}) \multimap (\Psi'; \overrightarrow{Fr}'; \tau)$ and $\Gamma \vdash (h; s) : (\Psi; \overrightarrow{Fr})$ hold. Then one of the following conditions holds:*

1. *$I = \boxed{v}$ for some value v.*
2. *There exists a small-step store ψ' and instruction I' such that $(h, s), I \leadsto \psi', I'$.*
3. *$I = E[\boxed{\ell}$ **unpack** $j]$ and $h(\ell) = pkg(\cdot)$.*

Proof. If I is a value, then condition 1 of the theorem holds immediately, proving the theorem. Assume I is not a value. Then I must have one of the following forms:

Case 1: $I = E[$**ldc.i4** $i4]$. Condition 2 holds with $I' = E[\boxed{i4}]$ and $\psi' = (h, s)$.

Case 2: $I = E[\boxed{v_1}; I_2]$. Condition 2 holds with $I' = E[I_2]$ and $\psi' = (h, s)$.

Case 3: $I = E[\boxed{v}\ I_2\ I_3\ \mathbf{cond}]$. By typing rule 37, $v = i4$ for some integer $i4$. Thus, condition 2 holds with $I' = E[I_j]$ and $\psi' = (h, s)$, where

$$j = \begin{cases} 3 & \text{if } i4 = 0 \\ 2 & \text{otherwise} \end{cases}$$

Case 4: $I = E[I_1\ I_2\ \mathbf{while}]$. Condition 2 holds with

$$I' = E[I_1\ (I_2; (I_1\ I_2\ \mathbf{while}))\ \boxed{\mathbf{0}}\ \mathbf{cond}]$$

and $\psi' = (h, s)$.

Case 5: $I = E[\mathbf{ldarg}\ j]$. By typing rule 42, $\overrightarrow{Fr} = \overrightarrow{Fr}_0(\tau_0, \ldots, \tau_n)$ and $0 \leq j \leq n$. Since $\Gamma \vdash_{stack} s : \overrightarrow{Fr}$, it follows from derivation rule 34 that $s = s_0(v_0, \ldots, v_n)$. Hence, condition 2 holds with $I' = E[\boxed{v_j}]$ and $\psi' = (h, s)$.

Case 6: $I = E[\boxed{v'}\ \mathbf{starg}\ j]$. By typing rule 43, $\overrightarrow{Fr} = \overrightarrow{Fr}_0(\tau_0, \ldots, \tau_n)$ and $0 \leq j \leq n$. Since $\Gamma \vdash_{stack} s : \overrightarrow{Fr}$, it follows from derivation rule 34 that $s = s_0(v_0, \ldots, v_n)$. Hence, condition 2 holds with $I' = E[\boxed{\mathbf{0}}]$ and $\psi' = (h, s_0(v_0, \ldots, v_{j-1}, v', v_{j+1}, \ldots, v_n))$.

Case 7: $I = E[\boxed{v_1}\ \ldots\ \boxed{v_n}\ \mathbf{newobj}\ C(\mu_1, \ldots, \mu_n)]$. Typing rule 44 implies that $n = \mathit{fields}(C)$. Condition therefore 2 holds with $I' = E[\boxed{\ell}]$ and $\psi' = (h[\ell \mapsto \mathit{obj}_C\{f_i \mapsto v_i | i \in 1..n\}^{\epsilon}], s)$.

Case 8: $I = E[\boxed{v_0}\ \ldots\ \boxed{v_n}\ \mathbf{callvirt}\ C{::}m.\mathit{Sig}]$. By typing rule 45, there exists I_0 such that $\mathit{methodbody}(C{::}m.\mathit{Sig}) = I_0$. Hence, condition 2 holds with $I' = E[I_0\ \mathbf{ret}]$ and $\psi' = (h, s(v_0, \ldots, v_n))$.

Case 9: $I = E[\boxed{v}\ \mathbf{ldfld}\ \mu\ C{::}f]$. By typing rule 40, $v = \ell$ such that $\Gamma \vdash \boxed{\ell} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; C\langle\ell\rangle)$ holds. Since $\Gamma \vdash_{heap} h : \Gamma$ holds, it follows from derivation rule 24 that $h(\ell) = \mathit{obj}_C\{\ldots, f = v, \ldots\}^{\vec{e}}$ for some value v. Hence, condition 2 holds with $I = E[\boxed{v}]$ and $\psi' = (h, s)$.

Case 10: $I = E[\boxed{v}\ \boxed{v'}\ \mathbf{stfld}\ \mu\ C{::}f]$. By typing rule 41, $v = \ell$ such that $\Gamma \vdash \boxed{\ell} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; C\langle\ell\rangle)$ holds. Since $\Gamma \vdash_{heap} h : \Gamma$ holds, it follows from derivation rule 24 that $h(\ell) = \mathit{obj}_C\{\ldots, f = v, \ldots\}^{\vec{e}}$ for some value v. Hence, condition 2 holds with $I = E[\boxed{\mathbf{0}}]$ and $\psi' = (h[\ell \mapsto \mathit{obj}_C[f \mapsto v']], s)$.

Case 11: $I = E[\boxed{v}\ \mathbf{evt}\ e_1]$. By typing rule 46, $v = \ell$ such that $\Gamma \vdash \boxed{\ell} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; C\langle\ell\rangle)$ holds. Since $\Gamma \vdash_{heap} h : \Gamma$ holds, it follows from derivation rule 24 that $h(\ell) = obj_C\{\ldots\}^{\overrightarrow{e}}$ for some event sequence $\overrightarrow{e}$. Thus, condition 2 of the theorem holds with $I' = E[\![\boxed{\mathbf{0}}]\!]$ and $\psi' = (h[\ell \mapsto obj_C\{\ldots\}^{\overrightarrow{e}e_1}, s)$.

Case 12: $I = E[\textbf{newpackage}\ C]$. Choose $\ell \notin Dom(h)$. Condition 2 holds with $I' = E[\![\boxed{\ell}]\!]$ and $\psi' = ((h, (\ell \mapsto pkg(\cdot))), s)$.

Case 13: $I = E[\boxed{v}\ \boxed{v'}\ \boxed{v''}\ \textbf{pack}]$. By typing rule 46, $v = \ell$ such that $\Gamma \vdash \boxed{\ell} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; C\langle ?\rangle)$ holds, $v' = \ell'$ for some heap pointer ℓ', and $v'' = \boldsymbol{rep}_C(H)$ for some history abstraction H. Since $\Gamma \vdash_{heap} h : \Gamma$ holds, it follows from derivation rule 25 that $h(\ell) = pkg(\ldots)$. Hence, condition 2 of the theorem holds with $I' = E[\![\boxed{\mathbf{0}}]\!]$ and $\psi' = (h[\ell \mapsto pkg(\ell', \boldsymbol{rep}_C(H))], s)$.

Case 14: $I = E[\boxed{v}\ \textbf{unpack}\ j]$. By typing rule 49, $v = \ell$ such that $\Gamma \vdash \boxed{\ell} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; C\langle ?\rangle)$ holds, and $\overrightarrow{Fr} = \overrightarrow{Fr}_0(\tau_0, \ldots, \tau_n)$ where $0 \leq j \leq n$. Since $\Gamma \vdash_{stack} s : \overrightarrow{Fr}$, it follows from derivation rule 34 that $s = s_0(v_0, \ldots, v_n)$. Since $\Gamma \vdash_{heap} h : \Gamma$, it follows from derivation rule 25 that $h(\ell) = pkg(\ldots)$. If $h(\ell) = pkg(\ell', v)$, then condition 2 of the theorem holds with $I = E[\![\boxed{\ell'}]\!]$ and $\psi' = (h[\ell \mapsto pkg(\cdot)], s_0(v_0, \ldots, v_{j-1}, v, v_{j+1}, \ldots, v_n))$. Otherwise $h(\ell) = pkg(\cdot)$ and therefore condition 3 of the theorem holds.

Case 15: $I = E[\![\boxed{v}\ I_2\ I_3\ \textbf{condst}\ k]$. By typing rule 50, $v = \boldsymbol{rep}_C(H)$ for some class C and history abstraction H. Condition 2 therefore holds with

$$I' = \begin{cases} E[I_3] & \text{if } test_k(C, \boldsymbol{rep}_C(H)) = 0 \\ E[I_2] & \text{otherwise} \end{cases}$$

and $\psi' = (h, s)$.

Case 16: $I = E[\![\boxed{v_1}\ \ldots\ \boxed{v_n}\ \textbf{newhist}\ k]$. By typing rule 51, $arity(HC_k) = n$. By axiom 22, it therefore follows that $arity(hc_k) = n$. Condition 2 of the theorem statement therefore holds with $I' = E[\![\boxed{hc_k(v_1, \ldots, v_n)}]\!]$ and $\psi' = (h, s)$.

□

B.4 Policy Adherence

The proof of Terminating Policy Adherence (Theorem 1) is as follows.

Proof. By subject reduction, there exists Γ', Ψ', $\overrightarrow{Fr}'$, and σ such that $\Gamma' \vdash \boxed{v} : (\Psi'; Fr') \multimap (\sigma(\Psi_{out}); \sigma(Fr_{out}); \sigma(\tau_{out}))$ and $\Gamma' \vdash (h'; s') : (\Psi'; Fr')$ hold.

From the typing rules for value expressions, we know that $\Psi' = \sigma(\Psi_{out})$ and $Fr' = \sigma(Fr_{out})$. Thus

$$\Gamma' \vdash (h'; s') : (\sigma(\Psi_{out}); \sigma(Fr_{out}); \sigma(\boxed{v})) \tag{62}$$

holds.

Let ℓ and $\overrightarrow{e}$ be given such that $h'(\ell) = obj_C\{\ldots\}^{\overrightarrow{e}}$. If $\ell \in Dom(\Psi_{out})$ then there exists a derivation of 62 with a subderivation of

$$\frac{\vdots \qquad\qquad \overrightarrow{e} \subseteq \sigma(\Psi_{out}(\ell))}{\vdash_{hist} h' : (\Gamma'; \Psi')}(28)$$

Since $\sigma(\Psi_{out}(\ell)) \subseteq policy(C)$ by assumption, we conclude that $\overrightarrow{e} \subseteq policy(C)$. If instead $\ell \notin Dom(\Psi_{out})$, then there exists a derivation of 62 with either a subderivation of

$$\frac{\vdots \qquad\qquad \overrightarrow{e} \subseteq \cdots \subseteq policy(C)}{\vdash_{hist} h' : (\Gamma'; \Psi')}(29)$$

or a subderivation of

$$\frac{\vdots \qquad\qquad \overrightarrow{e} \subseteq policy(C)}{\vdash_{hist} h' : (\Gamma'; \Psi')}(30)$$

In either case, we conclude that $\overrightarrow{e} \subseteq policy(C)$, satsifying the theorem. □

The proof of Non-terminating Prefix Adherence (Theorem 2) is as follows.

Proof. Proof is by induction on n.

Base Case: If $n = 0$ then $h' = h$ and the theorem holds by assumption.

Inductive Case: If $n \geq 1$ then there exists h_1, s_1, and I_1 such that $(h, s), I \rightsquigarrow^{n-1} (h_1, s_1), I_1$ holds and $(h_1, s_1), I_1 \rightsquigarrow (h', s'), I'$ holds. By inductive hypothesis, h_1 is prefix-adherent. By subject reduction, there also exists Γ_1, Ψ_1, Fr_1, and σ such that $\Gamma_1 \vdash I_1 : (\Psi_1; Fr_1) \multimap (\sigma(\Psi'); \sigma(\overrightarrow{Fr}'); \sigma(\tau))$ and $\Gamma_1 \vdash (h_1; s_1) : (\Psi_1; \overrightarrow{Fr}_1)$ hold.

Suppose $I_1 = E[\boxed{v_1} \ \ldots \ \boxed{v_n}\ \textbf{newobj}\ C(\mu_1, \ldots, \mu_m)]$. Then $h' = h, (\ell \mapsto obj_C\{\ldots\}^{\epsilon})$. Typing rule 44 implies that $\epsilon \in pre(policy(C))$. Since h is prefix-adherent, we conclude that h' is also prefix-adherent.

Suppose $I_1 = E[\boxed{\ell}\ \textbf{evt}\ e_1]$. Then h and h' are identical except for the event history of class object $h(\ell) = obj_C\{\ldots\}^{\overrightarrow{e}}$. Typing rule 46 implies that $\Psi_1(\ell)e_1 \subseteq pre(policy(C))$. Since $\ell \in Dom(\Psi_1)$ there exists a derivation of $\Gamma_1 \vdash (h_1; s_1) : (\Psi_1; \overrightarrow{Fr}_1)$ that includes a subderivation of

$$\frac{\vdots \qquad\qquad \overrightarrow{e} \subseteq \Psi_1(\ell)}{\vdash_{hist} (h_1; s_1) : (\Gamma_1; \Psi_1)}(28)$$

Thus, $\overrightarrow{e} e_1 \subseteq \mathit{pre}(\mathit{policy}(C))$, and we conclude that h' is prefix-adherent.

If I_1 has any other form, then h and h' are identical with respect to the event histories of their class objects. Since h is prefix-adherent by assumption, it follows that h' is prefix-adherent.

□