SECURITY POLICY ENFORCEMENT BY AUTOMATED PROGRAM-REWRITING

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Kevin W. Hamlen

August 2006

SECURITY POLICY ENFORCEMENT BY AUTOMATED PROGRAM-REWRITING

Kevin W. Hamlen, Ph.D.

Cornell University 2006

Traditional approaches to protecting computer systems from malicious or other misbehaved code typically involve (1) monitoring code for unacceptable behavior as it runs, or (2) detecting potentially misbehaved code and preventing it from running at all. These approaches are effective when unacceptable behavior can be detected in time to take remedial action, but in many settings and for many important security policies this is computationally expensive or provably impossible.

A third approach, termed in this dissertation *program-rewriting*, involves automatically rewriting code prior to running it in such a way that acceptable behavior is preserved but unacceptable behavior is not. Rewritten code can be run without further analysis or monitoring because it is guaranteed to exhibit only acceptable behavior. Program-rewriting has received recent attention in the literature in the form of *in-lined reference monitors*, which implement approach 1 above by in-lining security checks directly into the code being monitored. Program-rewriting generalizes in-lined reference monitoring, encompassing many other strategies for automatically rewriting programs as well.

This dissertation provides a formal characterization of the class of security policies enforceable by program-rewriting and shows that it is strictly greater than the classes of policies enforceable by all other known approaches combined. The disser-

tation also presents the design and implementation of a *certified program-rewriting* system for the .NET Common Language Infrastructure, showing that program-rewriters can be developed for real architectures. The extra step of certification provides a formal proof that the program-rewriter produces code that satisfies the security policy, resulting in additional guarantees that the implementation is correct, and higher levels of assurance.

BIOGRAPHICAL SKETCH

Kevin William Hamlen was born on June 2, 1976, in the suburbs of Buffalo, to parents William and Susan Hamlen. Being faculty members of the nearby state university, Kevin's parents raised their son with a love of learning, intellectual exploration, and academics. When he was seven years old, they purchased the family's first computer system—a Commodore™ 64 personal computer. Unexcited by the video games that came with it, Kevin soon picked up the system user manual, and it wasn't long before he was writing his own simple video games (in 6502 assembly language, because BASIC was too slow) and giving them to his friends to play.

Kevin's interest in computers continued through his junior year of high school, when he entered and won the 1993 national SuperQuest Supercomputing Competition and was awarded a summer at the Cornell Theory Center's supercomputing cluster to learn about parallel computing and distributed systems. The experience opened up a new world to the young student in which computers could be a science rather than merely a hobby, and it inspired him to pursue computer science as a profession.

In the following year he applied and was admitted to the Computer Science department of Carnegie Mellon University. During his four years there he drank in the lectures of many gifted researchers including Herb Simon, Mark Stehlik, Frank Pfenning, and Peter Lee. Inspired by Peter Lee's compiler design course, Kevin completed a senior honor's thesis with him and with George Necula on *Proof-Carrying Code for x86 Architectures* [Ham98], which won the Allen Newell award for excellence in undergraduate research.

In 1998 Kevin graduated from Carnegie Mellon with a bachelor's degree in Computer Science and Mathematical Sciences, and matriculated to Cornell University for his graduate education. There, professors Greg Morrisett and Fred Schneider introduced him to the emerging field of language-based security, where his long-standing interests in assembly languages, compilers, and abstract logic found root. After internships with Microsoft Research in 2001 and 2002, he completed a Master's of Science and Doctor of Philosophy degrees by designing and implementing an in-lined reference monitoring system for the Microsoft .NET Framework.

In the Fall of 2006, Kevin will continue his academic career by beginning work as an Assistant Professor for the Computer Science department at the University of Texas at Dallas.

For my wife, Rebecca,

whose enduring love and support

I will always cherish.

ACKNOWLEDGEMENTS

The author is greatly indebted to his major faculty advisors, Greg Morrisett and Fred B. Schneider, for their many contributions to this dissertation and for their patient and insightful tutelage in art of paper writing and computer science research. He also wishes to thank his minor advisor, Shimon Edelman, for many brilliant lectures on Cognition and the Psychological Sciences, which inspired this author to improve his own lecturing style and seek cross-disciplinary applications to this and future research.

The author extends his heartfelt thanks to colleague and longtime friend James Cheney, who endured (and possibly instigated) many late-night ramblings on computer science and mathematics. The presentation of material in this dissertation was also much improved by the comments of David Walker, Philip Fong, Tomás Uribe, Úlfar Erlingsson, Matthew Fluet, Yanling Wang, Michael Hicks, and Amal Ahmed, as well as anonymous referees for *TOPLAS* and *PLAS*.

The research in this dissertation was supported in part by AFOSR grants F49620-00-1-0198 and F49620-03-1-0156; Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory Air Force Material Command USAF under agreement number F30602-99-1-0533; National Science Foundation Grants 9703470, 0430161, and CCF-0424422 (TRUST); ONR Grant N00014-01-1-0968; and a grant from Intel Corporation.

More personally, the author greatly benefited throughout his graduate years from the warm companionship of his friends in the Graduate Christian Fellowship at Cornell and the Cornell International Christian Fellowship, including Rachel Blackwood, Amy Schlueter, Sara Sywulka, Dan Johnson, Kwang Taik Kim, and Jason Slinker.

Last and most significantly, the author expresses his love and appreciation for his wife, Rebecca, who tirelessly supported him and encouraged him throughout the preparation and writing of this dissertation, and for his God, from whom all wisdom and knowledge flow.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

Chapter 1

Introduction

1.1 Security Policy Enforcement

Modern software must balance facilitation of desired behavior (features) with prohibition of unacceptable behavior. With the rise of distributed and extensible systems, meeting this requirement has become more critical than it was in the past. Extensible systems, such as web browsers which download and run applet programs, or operating systems which incorporate drivers for new devices, must ensure that their extensions behave in a manner consistent with the intentions of the system designer and its users. When unacceptable behavior goes unchecked, damage can result—not only to the system itself but also to connected systems. Finding ways to reliably constrain system behavior is therefore important for protecting today's highly interconnected computing infrastructures.

The partitioning of possible system behavior into acceptable and unacceptable behavior constitutes a *security policy*. Acceptable behaviors are said to *satisfy* the security policy; unacceptable behaviors are said to *violate* the policy. If the notion of a system's behavior is broad enough to include not only the current state of the system but also the history of its past states as well as its potential future states, security policies include

- *access control* policies, such as a policy that restricts file access to users with appropriate credentials,
- *availability* policies, such as a policy that requires a web server to respond in a timely manner to each request for a web page, and

- *information flow* policies, such as a policy that prohibits user passwords from being leaked to other users.

Security enforcement mechanisms are employed to prevent unacceptable behavior. A security enforcement mechanism is said to (precisely) *enforce* a security policy when it permits all behavior that the policy classifies as acceptable and prevents all behavior that it classifies as unacceptable. A security enforcement mechanism that *conservatively enforces* a security policy by preventing both unacceptable behavior and some acceptable behavior can always be viewed as precisely enforcing a different, more restrictive, security policy. This precise view of enforcement is often more useful than conservative enforcement because identifying the security policy that a mechanism precisely enforces reveals the set of all security policies that it conservatively enforces along with a measure of how conservatively it enforces each of those policies. That is, it reveals both the unacceptable behaviors that the mechanism prevents as well as the acceptable behaviors that the mechanism might also prevent. This is important for identifying and excluding enforcement mechanisms that are overly-conservative in their enforcement. For example, an enforcement mechanism that prohibits all system behavior can be said to conservatively enforce all security policies, but is not terribly useful in practice, because it permits no desired behavior. To facilitate these kinds of analyses, I will henceforth reserve the term "enforcement" to refer only to precise enforcement.

Ideally, security enforcement mechanisms should enforce the policy that permits all behavior that is required in order to accomplish what the user and system administrators desire, and no other behavior. This idealized security policy instantiates the *principle of least privilege* [SS75], which asserts that every program should operate using the least set of privileges necessary to accomplish its function.

However, enforcing the principle of least privilege is usually not feasible in practice. For example, an investment program intended to earn its user a profit need only sell stocks at a gain, but enforcing a policy that admits only this behavior would require predicting the future of the stock market.

In practice, therefore, enforcement mechanisms must usually enforce some security policy that approximates the security policy of interest as closely as possible. This approximation usually involves conservatively prohibiting some behavior that might be acceptable. Developing enforcement mechanisms that enforce a wider range of richer and more expressive security policies is therefore advantageous because it allows the enforcement mechanism to more closely approximate security policies of interest, and to strike a better balance between prohibition of undesired behaviors and facilitation of desired behaviors.

Security enforcement mechanisms developed to date can be divided into three different classes based on their approach to enforcing security policies:

- *Static analyses* determine a program's possible behavior prior to its execution, rejecting some programs whose set of possible behaviors includes unacceptable behavior. For example, the static type-checking performed by the Java Virtual Machine [LY99, Chapter 4.9.1, Passes 1–3] is a static analysis that rejects malformed Java bytecode programs.

- *Execution Monitors* (EM's) [Sch00] dynamically monitor a program's behavior as it executes, intervening when an imminent policy violation is detected. For example, most operating systems implement access control matrices [Lam71], which guard access to system resources and which might prevent an illegal access by prematurely halting the offending program.

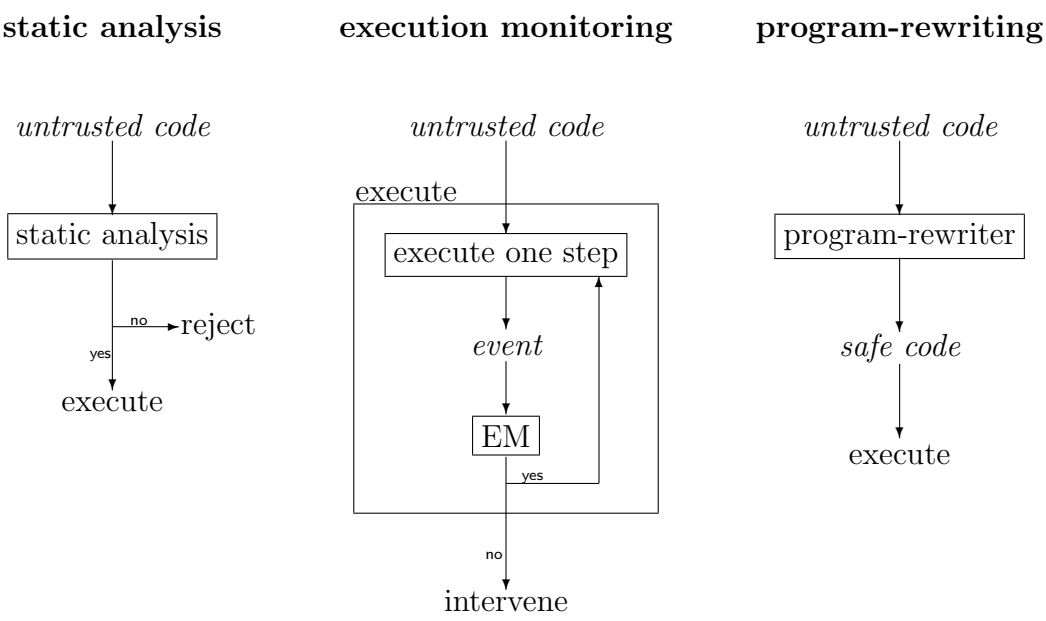


Figure 1.1: Load paths for static analyses, EM's, and program-rewriters

- *Program-rewriters* transform programs (some of which might be policy-violating) into policy-satisfying programs prior to their execution. For example, the SASI-Java system [ES00] rewrites Java programs by adding runtime security checks that halt the program if it would have otherwise violated the security policy.

Figure 1.1 compares typical load paths on systems that employ static analyses, EM's, and program-rewriters to enforce security policies. Observe that while static analyses accept or reject *untrusted code* prior to its execution, EM's observe security-relevant *events* that are *exhibited* by untrusted code during its execution, accepting or rejecting the code at runtime based on each such observation. When EM's maintain persistent internal state between observations, they can accept or reject based on the history of events observed to date. In this way, EM's can enforce a rich class of history-based security policies.

Program-rewriting can potentially combine the power of both static analyses and EM's by transforming untrusted code into *self-monitoring code* that performs security checks as it executes. They can additionally perform more-sophisticated transformations that allow the program to avoid potentially dangerous operations before they arise.

My Thesis. This dissertation champions program-rewriting as a powerful and effective means of enforcing security policies over untrusted code. It establishes that program-rewriting is the most powerful method of enforcing security policies currently known, it demonstrates that program-rewriting frameworks can be effectively implemented for real-world systems, and it presents a design strategy called *certified program-rewriting* whereby program-rewriters can be developed with high assurance that the implementation correctly protects against all policy violations.

1.2 Program-rewriting

A program-rewriter enforces security policies by accepting untrusted code as input and automatically transforming it into policy-adherent code. In order to satisfy the definition of policy enforcement, a program-rewriter must satisfy two requirements:

1. A program-rewriter is *sound* with respect to a security policy if it always produces code that satisfies the security policy.

2. A program-rewriter is *complete* with respect to a security policy if it preserves the behavior of code that already satisfies the security policy.

Program-rewriters that are sound protect against policy violations. Program-rewriters that are complete are not overly-conservative since that might prohibit

desired behavior. That is, a complete program-rewriter should not inhibit or change the behavior of a program that already complies with the security policy.

One example of program-rewriting is in-lined reference monitoring. An *In-lined Reference Monitor (IRM)* is an EM that has been implemented by injecting the dynamic security checks performed by the EM into the code being monitored. For example, consider an EM and an IRM that both enforce a *memory safety* security policy to prohibit programs from performing operations that access memory locations outside its address space. One example of an EM that enforces this policy is the memory management hardware of a computer system. Such an EM enforces this policy by monitoring each memory access and signaling an interrupt if the accessed location lies outside the bounds of the process's address space. A program-rewriter could enforce this policy by inserting bounds checks into untrusted code just before every program instruction that accesses memory. The result would be a IRM that halts itself before any illegal memory access would have otherwise occurred.

IRM implementations often have several advantages over EM's. First, security checks performed by IRM's can be more efficient than those performed by EM's because injecting security checks into the untrusted code itself avoids the overhead often imposed by context switches between the monitored code and the EM. For example, an EM implemented in an operating system must context switch from the user code being monitored to the operating system whenever a security check is performed. An IRM enforcing the same policy requires no such context switches since the security checks are performed by instructions within the user code's address space.

Second, IRM's can often avoid performing unnecessary security checks because the program-rewriter can analyze the untrusted code prior to executing it and avoid inserting unnecessary checks. For example, memory subsystems must usually check every memory access performed by user code, but IRM's can avoid checking at runtime those memory accesses that they can statically prove will always fall within the bounds of the code's address space. IRM's have been developed for several architectures that enforce memory safety and that use such techniques to achieve performance gains [WLAG93, Sma97, ABEL05, MM06].

Third, since an IRM's code lies within the untrusted code's address space, it can often take advantage of information not readily available to an EM that exists separately from the untrusted code. For example, a policy that requires a Java program to call its own class methods according to a prescribed protocol would be difficult to enforce by an EM implemented outside the Java program itself, since internal method calls are not usually observable to external processes.

Program-rewriters are not limited to implementing IRM's, however. Program-rewriters transform entire programs prior to executing them, whereas EM's and IRM's typically constrain their attention to individual *executions* or individual events in isolation, taking remedial action only if that individual execution or event constitutes a security policy violation. For example, information flow policies, such as the policy that requires that values of high-security variables not correlate with values of low-security variables across different executions of a given program, is not easily implementable by an EM or IRM that can only track the history of a program's current execution to date. A program-rewriter, however, can consider the set of all possible executions that a program might exhibit, trans-

forming untrusted code in sophisticated ways that break correlations between high- and low-security variables in order to enforce such information-flow policies.

The intuition that program-rewriters can enforce a strictly larger class of policies than those enforceable by EM's is proved formally in Chapter 2.

1.3 Certified Program-rewriting

Although program-rewriters are both powerful and efficient, they can be difficult to implement correctly. A production-level program-rewriter must be able to correctly analyze and transform a large domain of potentially complex untrusted code, it must perform these transformations quickly in order to minimize load time overhead, and it must produce efficient code so that transformed code executes at a speed that reasonably approximates the original untrusted code.

Program-rewriters that accomplish all these tasks well tend to be large and sophisticated pieces of software. They can therefore constitute a significant addition to a system's trusted computing base. Furthermore, as additional security policies of interest arise, it is convenient to develop specialized program-rewriters to enforce them. Thus, the growing pool of program-rewriters tends to enlarge a system's trusted computing base over time when every program-rewriter must be trusted.

Additions to a system's trusted computing base are tolerable when they come with formal guarantees of correctness, but program-rewriters developed to date have mostly lacked such guarantees. To achieve higher levels of assurance in mission-critical systems, it is desirable to have means of formally verifying the trusted components of the system to ensure that they satisfy and enforce the security policies desired by the system's users and administrators. No such means

has been developed for program-rewriting systems thus far (to my knowledge). It has only recently become possible to produce formal proofs of correctness for translations performed by realistic compilers [Ler06, BDL06]. Analogous proofs for realistic program-rewriters might be far off.

Language-based approaches to security [SMH01] have offered a number of powerful strategies for addressing these concerns by leveraging technologies from programming languages and compilers to develop robust, high-assurance security enforcement mechanisms. One such strategy is the advent of *low-level type systems*.

Low-level type systems allow code produced by a large piece of software (e.g., a compiler) to be formally verified by a smaller piece of software—a *type-checker*. This is accomplished by requiring the code-producer to generate code that includes typing annotations that suffice to prove that the code satisfies various desired properties. The type-checker verifies that the typing annotations prove that the code satisfies the property, thereby verifying that the code is safe to execute without trusting the code-producer, the code that it produced, or the typing annotations.

For example, Java compilers and .NET compilers produce code in a type-safe bytecode language (Java bytecode [LY99] and Common Intermediate Language (CIL) [ECM02], respectively). Code that is well-typed according to the language's type system is guaranteed to satisfy important invariants like memory safety and *control-flow safety*, which dictate that programs must access and transfer control only to certain suitable memory addresses throughout their executions. Bytecode verifiers for these languages can then check that the code produced by these compilers is well-typed, thereby proving that the code satisfies these invariants without trusting the compilers that produced the code. Proof-Carrying Code (PCC) [NL98] generalizes this approach by expressing the annotations as first-order logical pred-

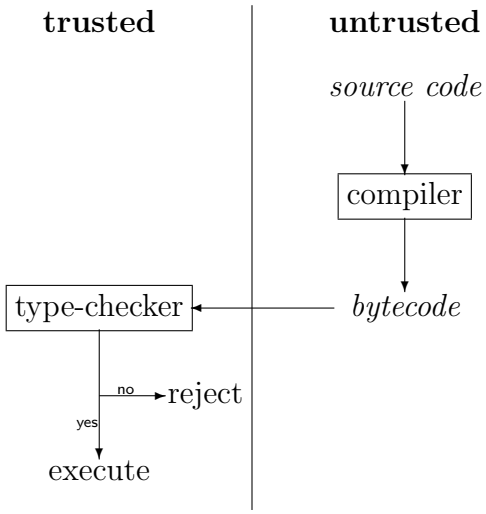


Figure 1.2: Bytecode type-checking

icates and proofs. A verifier for a PCC system checks that the annotations constitute a proof that the code satisfies the desired security policy. Figure 1.2 depicts the load path on a system that employs a type-safe bytecode language.

Chapter 3 proposes a low-level type system called *Mobile* which extends the .NET CIL type system in such a way that a Mobile type-checker can formally verify that CIL code produced by a program-rewriter satisfies a prescribed security policy. Such a type-system allows the development of certified program-rewriting systems—systems in which the program-rewriter itself can remain untrusted because a (smaller) type-checker can verify that the code produced by the program-rewriter satisfies the original security policy.

Figure 1.3 summarizes a typical load path on a system that executes code written in Mobile. Untrusted, managed, CIL code is first automatically rewritten according to a security policy, yielding an annotated, self-monitoring program written in Mobile. The rewriting can be performed by either a code producer or by a client machine receiving the untrusted code. Since the rewriter, and therefore

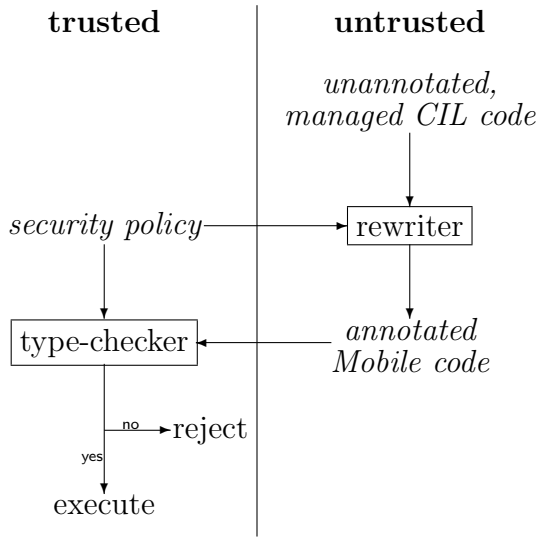


Figure 1.3: A Mobile load path

the self-monitoring program, remains untrusted, the self-monitoring program is then passed to a trusted type-checker that certifies the code with respect to the original security policy. Code that satisfies the security policy will be approved by the type-checker, and it is therefore safe to execute; code that is not well-typed will be rejected, indicating a failure of the rewriter.

Mobile programs can be implemented as CIL binaries that use CIL custom attributes to store typing annotations. This means that certified program-rewriting systems can be developed for .NET architectures without modifying the .NET virtual machine, compiler, or runtime system. Chapter 4 discusses the development and implementation of a certified program-rewriting system that uses Mobile to implement certified IRM's for the Microsoft .NET Framework.

1.4 Specifying Security Policies

Although many real-world systems today have security policies that exist only as informal, implicit notions in the minds of the system's administrators, security policies of interest should be specified explicitly by writing them in a declarative language whenever the security of the system is important to its users and maintainers. Explicit policy specifications permit formal reasoning about the security policy, which facilitates (i) assessing whether a given security policy is correct in the sense that it accurately represents the set of behaviors desired by the system's users and maintainers, and (ii) determining whether a given system has possible behaviors that violate the security policy.

In a certified program-rewriting system, explicit policy specifications are additionally useful in that they permit automatic verification of the soundness of the program-rewriter. By passing the same policy specification as input to both the program-rewriter and the type-checker, the type-checker can independently verify that the program-rewriter has produced code that satisfies the policy.

Writing policy specifications declaratively also achieves a useful separation of concerns in a certified program-rewriting system. Program-rewriters often have a wide range of possible ways to enforce any given security policy. For example, an IRM that detects an imminent policy violation could intervene by (a) halting the program, (b) skipping the policy-violating operation and continuing execution, (c) throwing a security exception that could be caught by surrounding code, or (d) executing some error-handling code provided by the policy writer. A program-rewriter that produces IRM's might accept advice from the system administrator about which of these error-handling methods to implement. Separating such imperative advice from the declarative policy specification allows a type-checker to

verify that any error-handling implemented by the program-rewriter does not itself cause a policy violation. Since the policy specification is a trusted component of the system, this reduces the trusted computing base by excluding any imperative advice. In this way the (declarative) security policy is distinguished from any (imperative) advice on *how* to enforce the security policy.

Security policy specifications in this dissertation will consist of three main components: (i) identification of the set of security-relevant *entities*, (ii) identification of security-relevant events on those entities, and (iii) declaration of which sequences of events are permissible and which are not.

Entities The entities in a security policy are the resources to which the policy restricts access. For example, a policy that restricts access to files might identify file handles as entities. In an object-oriented setting, entities tend to be objects. For example, in Mobile, entities are .NET objects as defined by the Common Language Infrastructure (CLI). Specifying a Mobile security policy requires identifying the set of security-relevant classes whose object instantiations should be monitored by the enforcement mechanism.

Events Events define the program operations that constitute access to an entity. For example, a security policy that restricts writing to files would identify as security-relevant events all the program instructions that cause files to be written. In object-oriented frameworks where entities are objects, events are often instance method calls on those objects. A policy written in Mobile could, for example, restrict `File.Write()` method calls if objects of class `File` are security-relevant entities.

Event Sequences Security policies specify the sequences of events that programs are permitted to exhibit at runtime. For example, a policy that prohibits access to a particular file would permit all event sequences that do not include an event that constitutes writing to that file. A resource bound policy might permit only those event sequences that include n or fewer events that constitute accesses to the resource, where n is some constant. Sets of permissible event sequences can be specified using a variety of paradigms including security automata [AS87, Sch00], regular expressions, or linear temporal logic [Eme90].

In settings where not every entity can be statically named, it is also useful to be able to specify event sequences that are quantified over sets of entities. For example, a policy might require that all opened files must eventually be closed. When the set of files on the system is unbounded, such a policy can be stated formally by requiring that, for each entity f of type file, every event $e_{open(f)}$ (corresponding to opening the file) that appears in the sequence must be eventually followed by an event $e_{close(f)}$ (corresponding to closing the file). Observe that the universal quantification allows the security policy to refer to all files without providing a name for each. Object-oriented settings provide a natural method of quantifying over objects by appealing to the class hierarchy. For example, Mobile policies can refer to all objects of class `File`.

Specifying information flow policies requires a fourth component beyond what is described above. An information flow policy specifies a set of permissible *sets* of event sequences rather than identifying individual sequences as acceptable or unacceptable. For example, a policy that prohibits programs from divulging a secret might require that the set of event sequences that any given program might

exhibit must be a singleton set. If the set of events was defined to be those operations that are outwardly observable, then this would force any given program to exhibit consistent observable behavior regardless of the value of the secret. Although Chapter 2 reasons about such policies at an abstract level, Mobile does not currently include support for information flow policies. This fourth level in formal policy specifications is thus omitted from this dissertation.

1.5 Structure of the Dissertation

Chapter 2 presents a formal framework for reasoning about security enforcement mechanisms and the classes of security policies they can enforce. It exposes flaws in previous work intended to characterize the class of policies enforceable by EM's, resulting in a more precise characterization of the power of EM's. It uses this characterization to show that program-rewriters can enforce a class of policies strictly greater than those enforceable by static analyses and EM's combined.

Chapter 3 shows how to remove program-rewriters from the trusted computing base by replacing them with a trusted type-checker. It presents Mobile, a type system that extends the .NET CIL type system to support certified program-rewriting. Formal proofs show that well-typed Mobile code is guaranteed to be policy adherent.

Chapter 4 then describes a prototype implementation of Mobile for the Microsoft .NET Framework. The implementation includes a program-rewriter that automatically transforms untrusted CIL code into annotated Mobile code in accordance with a declarative security policy, along with a type-checker that verifies that annotated Mobile code is well-typed with respect to a security policy. Prelim-

inary tests are presented that indicate that the approach constitutes an effective and efficient means of enforcing a rich class of history-based security policies.

Chapter 2

Computational Power of Security Enforcement Mechanisms

The material in this chapter is derived from previously published [HMS06b] joint work with Greg Morrisett and Fred B. Schneider.

2.1 Overview

Assessing the power of program-rewriting relative to other security enforcement strategies requires formally characterizing each strategy and identifying the classes of security policies enforceable by a mechanism that employs each. Such work is useful because it allows us to assess the power of different mechanisms, choose mechanisms well suited to particular security needs, identify kinds of attacks that might still succeed even after a given mechanism has been deployed, and derive meaningful completeness results for newly developed mechanisms.

Comparing enforcement methods in this way requires an abstract model of security policies and security policy enforcement that is broad enough to cover all policies and mechanisms of interest. In this dissertation we adopt the model proposed by Schneider [Sch00] wherein program behavior is regarded as a sequence of events. This model is sufficiently flexible to capture security policies that concern a program's history of operations rather than merely individual program operations. We further adopt Viswanathan's view [Vis00] that, in practice, both enforcement mechanisms and the untrusted code they constrain are subject to computability constraints.

Prior work [Sch00, Vis00] has characterized a class EM_{orig} of security policies meant to capture what could be effectively enforced through execution monitoring. Program-rewriting, however, can be viewed as a generalization of execution monitoring. No characterization of the class of security policies enforceable by program-rewriting has been previously developed (to our knowledge). Since numerous systems [DG71, WLAG93, Sma97, ES99, ET99, ES00] use program-rewriting in ways that go beyond what can be modeled as an EM, a characterization of the class of policies enforceable by program-rewriters would be useful. In this chapter we extend the model proposed by [Sch00] to characterize this new class of policies, the *RW-enforceable* policies, corresponding to what can be effectively enforced through program-rewriting.

Execution monitoring can be viewed as an instance of program-rewriting, so one would expect class EM_{orig} to be a subclass of the RW-enforceable policies. However, we show that surprisingly this is not the case; there are some policies in EM_{orig} that are not enforceable by any program-rewriter. Our analysis of these policies shows that they cannot actually be enforced by an execution monitor either, revealing that EM_{orig} actually constitutes an upper bound on the class of policies enforceable by execution monitors instead of an exact bound as was previously thought. We then show that intersecting EM_{orig} with the RW-enforceable policies yields exactly those policies that can actually be enforced by an execution monitor, the *EM-enforceable* policies.

The chapter proceeds as follows. A formal model of security enforcement is defined in §2.2. Next, in §2.3 that model is used to characterize and relate three methods of security enforcement: static analysis, execution monitoring, and program-rewriting. Using the results of these analyses, §2.4 exposes and corrects flaws in

prior work that cause EM_{orig} to admit policies not enforceable by any execution monitor. Related work is discussed in §2.5 and future work is discussed in §2.6. Finally, §2.7 summarizes the results of the prior sections.

2.2 Formal Model of Security Enforcement

2.2.1 Programs and Executions

An enforcement mechanism prevents unacceptable behavior by *untrusted programs*. Fundamental limits on what an enforcement mechanism can prevent arise whenever that mechanism is built using computational systems no more powerful than the systems upon which the untrusted programs themselves are based, because the incompleteness results of Gödel [Göd31] and Turing [Tur36] then imply there will be questions about untrusted programs unanswerable by the enforcement mechanism.

To expose these unanswerable questions, untrusted programs must be represented using some model of computation. The Turing Machine (TM) [Tur36] is an obvious candidate because it is well understood and because a wide range of security policies can be encoded as properties of Turing Machines [Mar89, HRU76].

Recall, a TM has a finite control comprising a set of states and a transition relation over those states. A *computational step* occurs whenever a TM moves from one finite control state to another (possibly the same) finite control state in accordance with its transition relation.

However, there are two reasons that the traditional definition of a TM, as a one-tape finite state machine that accepts or rejects finite-length input strings, is unsuitable for representing untrusted programs. First, it does not model non-terminating programs well. Operating systems, which are programs intended to

run indefinitely, are not easily characterized in terms of acceptance or rejection of a finite input. Second, the traditional definition of a TM does not easily distinguish runtime information that is observable by the outside world (e.g. by an enforcement mechanism) from runtime information that is not observable because all runtime information is typically encoded on a single tape. Any realistic model of execution monitoring must be rich enough to express the monitor's limited power to access some but not all information about the untrusted program as it runs.

Therefore, untrusted programs are modeled in this chapter by a multitape variant of a traditional TM (multitape TM's being equivalent in computational power to single-tape TM's [HU79, p. 161]) that we term a *program machine* (PM). PM's are deterministic TM's (i.e. TM's with deterministic transition relations) that manipulate three infinite-length tapes:

- An *input tape*, which contains information initially unavailable to the enforcement mechanism: user input, non-deterministic choice outcomes, and any other information that only becomes available to the program as its execution progresses. Non-determinism in an untrusted program is modeled by using the input tape contents, even though PM's are themselves deterministic. Input tapes may contain any finite or infinite string over some fixed, finite alphabet Γ; the set of all (finite-length and infinite-length) input strings is denoted by Γ^{ω}.

- A *work tape*, which is initially blank and can be read or written by the PM without restriction. It models work space provided to the program at runtime and is not directly available to the enforcement mechanism.

- A write-only *trace tape*, discussed more thoroughly below, which records security-relevant behavior by the PM that can be observed by the enforcement mechanism.

Separation of a PM's runtime information into these three tapes allows us to provide PM's infinite-length input strings on their input tapes, and allows the model to distinguish information that is available to the enforcement mechanism from information that is not.

As a PM runs, it exhibits a sequence of events observable to the enforcement mechanism by writing encodings of those events on its trace tape. For example, if "the PM writing a 1 to its work tape" is an event that the enforcement mechanism will observe, and the encoding of this event is "0001", then the string "0001" is automatically written by the PM to its trace tape whenever that PM writes a 1 to its work tape.

As the example suggests, we assume a fixed universe of all observable events E and assume that their encodings do not vary from PM to PM. Assuming a fixed set E allows our model to distinguish between information observable by the enforcement mechanism and information that is not observable. It can be used to specify that some information might never be available to an enforcement mechanism and that other information, like user inputs or non-deterministic choices, only becomes available to the enforcement mechanism at a particular point during execution. The result is a model that distinguishes between two different (but often conflated) reasons among the many reasons why an enforcement mechanism might fail to enforce a particular security policy:

- The enforcement mechanism could fail because it lacks the ability to observe events critical to the enforcement of the policy. In that case, E is inadequate to enforce the policy no matter which enforcement mechanism is employed.

- The enforcement mechanism could fail because it lacks sufficient computational power to prevent a policy violation given the available information. In this case, where one enforcement mechanism fails, another might succeed.

Although we do not fix a specific set E, we will make several assumptions about E that allow us to model the predictive power of enforcement mechanisms of interest in this chapter. In particular, enforcement mechanisms that seek to prevent security policy violations before they occur must always have some ability to predict an untrusted program's behavior finitely far into the future on any given input. For example, execution monitors must be able to look ahead at least one computational step on all possible control flow paths to see if a security-relevant event might next be exhibited. Without this ability, they have no opportunity to intercept bad events before they occur. This predictive power is, of course, limited by the information available to the enforcement mechanism. An enforcement mechanism cannot necessarily determine which (if any) event will be exhibited next if the untrusted program is about to read input, but we will assume that it can determine which event would be exhibited next for any given input symbol the untrusted program might read. This prediction will be repeatable for a finite number of iterations to predict the outcome of any given finite sequence of inputs that the untrusted program might encounter. The following assumptions about E suffice to model this predictive power.

- E is a countably infinite set, allowing each event to be unambiguously encoded as a finite sequence of symbols on a PM's trace tape.

- Reading a symbol from the input tape is always an observable event. Thus for each input symbol $s \in \Gamma$, there is an event $e_s \in E$ that corresponds to reading s from the input tape.

- For each PM M, there is an event e_M that encodes M, including the finite control and transition relation of M.[1] This corresponds to the assumption that the enforcement mechanism can access the untrusted program's text to make finite predictions about its future behavior. A means for the enforcement mechanism to use event e_M to make these predictions will be given shortly.

A weaker set of assumptions about E that permits enforcement mechanisms access to less information, but that still captures the predictive power of interesting enforcement mechanisms might be possible but is left as future work. Independent work on this problem [Fon04] is discussed in §2.5.

Following Schneider [Sch00], program executions are modeled as sequences χ of events from E. Without loss of generality, we assume that complete executions are always infinite event sequences. (If an untrusted program's termination is an observable event, then it can be modeled by a PM that loops, repeatedly exhibiting a distinguished event e_{end}, instead of terminating. If program termination is not observable, E can be augmented with an event e_{skip} that indicates that either the untrusted program has terminated or that no security-relevant event has taken

[1] This assumption might appear to give an enforcement mechanism arbitrarily powerful decision-making ability, but we will see in §2.3 that the power is still quite limited because unrestricted access to the program text is tempered by time limits on the use of that information.

place on a particular computational step.) Many of our analyses will involve both complete executions and their finite prefixes, and we use χ to refer to both infinite and finite event sequences unless explicitly stated otherwise. Each finite prefix of an execution encodes the information available to the enforcement mechanism up to that point in the execution. Define $|\cdot| : E^\omega \to (\mathbb{N} \cup \{\infty\})$ such that $|\chi|$ is the length of sequence χ if χ is finite, and $|\chi|$ is ∞ if χ is infinite. When $i \in \mathbb{N}$ and $1 \leq i \leq |\chi|$ then let $\chi[i]$ denote the ith event of χ, let $\chi[..i]$ denote the length-i prefix of χ, and let $\chi[i+1..]$ denote the suffix of χ consisting of all but the first i events.

Executions exhibited by a PM are recorded on the PM's trace tape. As the PM runs, a sequence of symbols gets written to the trace tape—one (finite) string of symbols for each event $e \in E$ the PM exhibits. If the PM terminates, then the encoding of e_{end} or e_{skip} is used to pad the remainder of the (infinite-length) trace tape. Let $\chi_{M(\sigma)}$ denote the execution written to the trace tape when PM M is run on input tape σ. Let X_M denote the set of all possible executions exhibited by a PM M (*viz* $\{\chi_{M(\sigma)} | \sigma \in \Gamma^\omega\}$), and let X_M^- denote the set of all non-empty finite prefixes of X_M (*viz* $\{\chi[..i] | \chi \in X_M, i \geq 1\}$).

To provide enforcement mechanisms with the ability to anticipate the possible exhibition of security-relevant events, we assume that the first event of every execution exhibited by M is event e_M. Thus, we assume that there exists a computable function $\langle\!\langle \cdot \rangle\!\rangle$ from executions to PM's such that $\langle\!\langle \chi_{M(\sigma)}[..i] \rangle\!\rangle = M$ for all $i \geq 1$.

Although the existence of a function $\langle\!\langle \cdot \rangle\!\rangle$ that maps executions to the PM's that generated them is a realistic assumption, in so far as enforcement mechanisms located in the processor or operating system have access to the program, only

superficial use has been made of this information in actual execution monitor implementations to date. For example, reference monitors in kernels typically do not perform significant analyses of the text of the untrusted programs they monitor. Instead they use hardware interrupts or other instrumentation techniques that consider only single program instructions in isolation. Eliminating from our formal model the assumption that $\langle\!\langle \cdot \rangle\!\rangle$ exists would require a model in which the computational details of PM's are more elaborate, because the model would need to allow enforcement mechanisms to examine enough of a PM to predict security-relevant events before they occur but not enough to recover all of the PM's finite control. We conjecture that such an elaboration of our model would result in only trivial changes to the results derived in this chapter, but a proper analysis is left to future work.

To ensure that a trace tape accurately records an execution, the usual operational semantics of TM's, which dictates how the finite control of an arbitrary machine behaves on an arbitrary input, is augmented with a fixed *trace mapping* $(M, \sigma) \mapsto \chi_{M(\sigma)}$ such that the trace tape unambiguously records the execution that results from running an arbitrary PM M on an arbitrary input σ.

Appendix A provides a formal operational semantics for PM's, an example event set, and an example trace mapping satisfying the constraints given above.

2.2.2 Security Policies

A security policy defines a binary partition on the set of all (computable) sets of executions. Each (computable) set of executions corresponds to a PM, so a security policy divides the set of all PM's into those that satisfy the policy and those that

do not. This definition of security policies is broad enough to express most things usually considered security policies [Sch00], including

- access control policies, which are defined in terms of a program's behavior on an arbitrary individual execution for an arbitrary finite period,
- availability policies, which are defined in terms of a program's behavior on an arbitrary individual execution over an infinite period, and
- information flow policies, which are defined in terms of the set of all executions—and not the individual executions in isolation—that a program could possibly exhibit.

Given a security policy $\mathcal{P}$, we write $\mathcal{P}(M)$ to denote that M satisfies the policy and $\neg\mathcal{P}(M)$ to denote that it does not.

For example, if cells 0 through 511 of the work tape model the boot sector of a hard disk, and we have defined E such that a PM exhibits event $e_{writes(i)} \in E$ whenever it writes to cell i of its work tape, then we might be interested in the security policy $\mathcal{P}_{boot}$ that is satisfied by exactly those PM's that never write to any of cells 0 through 511 of the work tape. More formally, $\mathcal{P}_{boot}(M)$ holds if and only if for all $\sigma \in \Gamma^\omega$, execution $\chi_{M(\sigma)}$ does not contain any of events $e_{writes(i)}$ for $0 \leq i < 512$.

Security policies are often specified in terms of individual executions they prohibit. Letting $\hat{\mathcal{P}}$ be a predicate over executions, the security policy $\mathcal{P}$ *induced by* $\hat{\mathcal{P}}$ is defined by:

$$\mathcal{P}(M) =_{\text{def}} (\forall \chi : \chi \in X_M : \hat{\mathcal{P}}(\chi))$$

That is, a PM M satisfies a security policy $\mathcal{P}$ if and only if all possible executions of M satisfy predicate $\hat{\mathcal{P}}$. $\hat{\mathcal{P}}$ is called a *detector* for $\mathcal{P}$. For example, if we define

a detector $\hat{\mathcal{P}}_{boot}(\chi)$ to hold exactly when χ does not contain any event $e_{writes(i)}$ for $0 \leq i < 512$, then policy $\mathcal{P}_{boot}$ (above) is the policy induced by detector $\hat{\mathcal{P}}_{boot}$.

The detector $\hat{\mathcal{P}}_{boot}$ can be decided[2] for a finite execution prefix χ by verifying that χ does not contain any of a set of *prohibited events*, namely $e_{writes(i)}$ for $0 \leq i < 512$. Such detectors are often useful, so for any set of events $B \subseteq E$ to be prohibited, we define[3]:

$$\hat{\mathcal{P}}_B(\chi) =_{\text{def}} (\forall e : e \in \chi : e \notin B)$$

The policy $\mathcal{P}_B$ induced by $\hat{\mathcal{P}}_B$ is satisfied by exactly those PM's that never exhibit an event from B. Policy $\mathcal{P}_{boot}$ can then be expressed as $\mathcal{P}_{\{e_{writes(i)}|0\leq i<512\}}$.

2.3 Modeling Various Security Enforcement Mechanisms

The framework defined in §2.2 can be used to model many security enforcement mechanisms, including static analyses (c.f. [LY99, Chapter 4.9.1, Passes 1–3], [Nac97, MCG99, Mye99]), reference monitors (c.f. [LY99, Chapter 4.9.1, Pass 4], [Lam71, And72, RC91, KKLS01]), and program-rewriters (c.f. [DG71, WLAG93, Sma97, ET99, ES00, BLW05, HMS06a]).

2.3.1 Static Analysis

Enforcement mechanisms that accept or reject an untrusted program strictly prior to running the untrusted program are termed *static analyses*. Here, the enforcement mechanism must determine whether the untrusted program satisfies the se-

[2]We say a predicate can be *decided* or is *recursively decidable* iff there exists an algorithm that, for any finite-length element, terminates and returns 1 if the element satisfies the predicate, and terminates and returns 0 otherwise.

[3]We write $e \in \chi$ holds if and only if event e is in execution χ; i.e. $e \in \chi =_{\text{def}} (\exists i : 0 \leq i : e = \chi[i])$.

curity policy within a finite period of time.[4] Accepted programs are permitted to run unhindered; rejected programs are not run at all. Examples of static analyses include static type-checkers for type-safe languages, like that of the Java Virtual Machine[5] [LY99, Chapter 4.9.1, Passes 1–3] and TAL [MCG99]. JFlow [Mye99] and others use static analyses to provide guarantees about other security policies such as information flow. Standard virus scanners [Nac97] also implement static analyses.

Formally, we deem a security policy to be statically enforceable in our model if it satisfies the following definition.

Definition 2.1. A security policy $\mathcal{P}$ is *statically enforceable* if there exists a TM $M_{\mathcal{P}}$ that takes an encoding of an arbitrary PM M as input and, if $\mathcal{P}(M)$ holds, then $M_{\mathcal{P}}(M)$ accepts in finite time; otherwise $M_{\mathcal{P}}(M)$ rejects in finite time.

Thus, by definition, statically enforceable security policies are the recursively decidable properties of TM's:

Theorem 2.1. *The class of statically enforceable security policies is the class of recursively decidable properties of programs (also known as class* Π_0 *of the arithmetic hierarchy).*

Proof. Immediate from Definition 2.1. Recursively decidable properties are, by definition, those for which there exists a total, computable procedure that decides them. Machine $M_{\mathcal{P}}$ is such a procedure. □

[4]Some enforcement mechanisms involve simulating the untrusted program and observing its behavior for a finite period. Even though this involves running the program, we still consider it a static analysis as long as it is guaranteed to terminate and yield a yes or no result in finite time.

[5]The JVM also includes runtime type-checking in addition to static type-checking. The runtime type-checks would not be considered to be static analyses.

Class Π_0 is well-studied, so there is a theoretical foundation for statically enforceable security policies. Statically enforceable policies include: "M terminates within 100 computational steps," "M has less than one million states in its finite control," and "M writes no output within the first 20 steps of computation." For example, since a PM could read at most the first 100 symbols of its input tape within the first 100 computational steps, and since Γ is finite, the first of the above policies could be decided in finite time for an arbitrary PM by simulating it on every length-100 input string for at most 100 computational steps to see if it terminates. Policies that are not statically enforceable include, "M eventually terminates," "M writes no output when given σ as input," and "M never terminates." None of these are recursively decidable for arbitrary PM's.

Static analyses can also prevent PM's from violating security policies that are not recursively decidable, but only by enforcing policies that are conservative approximations of those policies. For example, a static analysis could prevent PM's from violating the policy, "M eventually terminates," by accepting only PM's that terminate within 100 computational steps. However, in doing so it would conservatively reject some PM's that satisfy the policy—specifically, those PM's that terminate after more then 100 computational steps.

2.3.2 Execution Monitoring

Execution Monitors (EM's), described in §1.1, include reference monitors [And72, War79] and other enforcement mechanisms that operate alongside an untrusted program. An EM intercepts security-relevant events exhibited as the untrusted program executes, and the EM intervenes upon seeing an event that would lead to a violation of the policy being enforced. The intervention might involve ter-

minating the untrusted program or might involve taking some other corrective action.[6] Examples of EM enforcement mechanisms include access control list and capability-based implementations of access control matrices [Lam71] as well as hardware support for memory protection. Runtime linking performed by the Java Virtual Machine [LY99, Chapter 4.9.1, Pass 4], and runtime type-checking such as that employed by dynamically typed languages like Scheme [RC91], are other examples. The MaC system [KKLS01] implements EM's through a combination of runtime event-checking and program instrumentation.

Schneider [Sch00] observes that for every EM-enforceable security policy $\mathcal{P}$ there will exist a detector $\hat{\mathcal{P}}$ such that

$$\mathcal{P}(M) \equiv (\forall \chi : \chi \in X_M : \hat{\mathcal{P}}(\chi)) \tag{EM1}$$

$$\hat{\mathcal{P}}(\chi) \Longrightarrow (\forall i : 1 \leq i < |\chi| : \hat{\mathcal{P}}(\chi[..i])) \tag{EM2}$$

$$\neg\hat{\mathcal{P}}(\chi) \Longrightarrow (\exists i : 1 \leq i : \neg\hat{\mathcal{P}}(\chi[..i])) \tag{EM3}$$

That is, detector $\hat{\mathcal{P}}$ induces policy $\mathcal{P}$, and executions satisfy detector $\hat{\mathcal{P}}$ if and only if all of their prefixes satisfy $\hat{\mathcal{P}}$. Historically, constraints EM1 – EM3 have been used to formally characterize the class of *safety properties* [ADS86]—properties that stipulate some "bad thing" does not happen during execution [Lam77]. This previous work also implicitly assumed an additional constraint on detectors, made explicit by Viswanathan [Vis00]:

$$\hat{\mathcal{P}}(\chi) \text{ is recursively decidable whenever } \chi \text{ is finite.} \tag{EM4}$$

[6] Schneider [Sch00] assumes the only intervention action available to an EM is termination of the untrusted program. Since we are concerned here with a characterization of what policies an EM can enforce, it becomes sensible to consider a larger set of interventions.

That is, detectors must be computable. We refer to the class of security policies given by EM1 – EM4 as class EM_{orig}. A security policy $\mathcal{P}$ in EM_{orig} can be enforced by deciding $\hat{\mathcal{P}}$ at each computational step. Specifically, as soon as the next exhibited event, if permitted, would yield an execution prefix that violates $\hat{\mathcal{P}}$, the EM intervenes to prohibit the event.

EM4 is critical for providing a formal definition that agrees with the informal notion of a safety property [Lam77], because EM4 rules out detectors that are arbitrarily powerful and thus not available to any real EM implementation. For example, choose a surjection $G : E \rightarrow PM$ from events to PM's[7] and define $L =_{\text{def}} \{e | G(e)(\epsilon) \text{ never halts}\}$ to be the set of events corresponding to PM's that never halt on input ϵ (the empty sequence). The policy $\mathcal{P}_L$ that requires a PM to exhibit only events corresponding to PM's that eventually halt—a liveness property that no EM can enforce [Lam77, Sch00]—satisifes EM1 – EM3 but not EM4. No EM can enforce policy $\mathcal{P}_L$ because there is no decision procedure that an EM can use to determine, for an arbitrary event e, whether PM $G(e)$ eventually halts.

In §2.4.1 we show that real EM's are limited by additional constraints. However, class EM_{orig} constitutes a useful upper bound on the set of policies enforceable by execution monitors. Viswanathan [Vis00] shows that EM_{orig} is equivalent to the co-recursively enumerable (coRE) properties, also known as class Π_1 of the arithmetic hierarchy. A security policy $\mathcal{P}$ is coRE when there exists a TM $M_{\mathcal{P}}$ that takes an arbitrary PM M as input and rejects it in finite time if $\neg\mathcal{P}(M)$ holds; otherwise $M_{\mathcal{P}}(M)$ loops forever. The equivalence of EM_{orig} to the coRE properties will be used throughout the remainder of the chapter. Since Viswanathan's model differs substantially from ours, we reprove this result for our model.

[7]Such a mapping exists because set E and the set of all program machines are both countably infinite.

Theorem 2.2. *The class given by EM1 – EM4 is the class of co-recursively enumerable (coRE) properties of programs (also known as the* Π_1 *class of the arithmetic hierarchy).*

Proof. First we show that every policy satisfying EM1 – EM4 is coRE. Let a policy $\mathcal{P}$ satisfying EM1 – EM4 be given. Security policy $\mathcal{P}$ is, by definition, coRE if there exists a TM $M_{\mathcal{P}}$ that takes an arbitrary PM M as input and loops forever if $\mathcal{P}(M)$ holds but otherwise halts in finite time. To prove that $\mathcal{P}$ is coRE, we construct such an $M_{\mathcal{P}}$.

By EM1, $\mathcal{P}(M) \equiv (\forall \chi : \chi \in X_M : \hat{\mathcal{P}}(\chi))$ for some $\hat{\mathcal{P}}$ satisfying EM2 – EM4. EM4 guarantees that a TM can decide $\hat{\mathcal{P}}(\chi)$ for finite χ. We can therefore construct $M_{\mathcal{P}}$, as follows: When given M as input, $M_{\mathcal{P}}$ begins to iterate through every finite prefix χ of executions in X_M. For each, it decides $\hat{\mathcal{P}}(\chi)$. If it finds a χ such that $\neg\hat{\mathcal{P}}(\chi)$ holds, it halts. (This is possible because EM4 guarantees that $\hat{\mathcal{P}}(\chi)$ is recursively decidable.) Otherwise it continues iterating indefinitely.

If $\mathcal{P}(M)$ holds, then by EM1, there is no $\chi \in X_M$ such that $\neg\hat{\mathcal{P}}(\chi)$ holds. Thus, by EM2, there is no i such that $\neg\hat{\mathcal{P}}(\chi[..i])$ holds. Therefore $M_{\mathcal{P}}$ will loop forever. But if $\mathcal{P}(M)$ does not hold, then by EM1 and EM3 there is some χ and some i such that $\neg\hat{\mathcal{P}}(\chi[..i])$ holds. Therefore $M_{\mathcal{P}}$ will eventually terminate. Thus, $M_{\mathcal{P}}$ is a witness to the fact that policy $\mathcal{P}$ is coRE.

Second, we show that every coRE security policy satisfies EM1 – EM4. Let a security policy $\mathcal{P}$ that is coRE be given. That is, assume there exists a TM $M_{\mathcal{P}}$ such that if $\mathcal{P}(M)$ holds then $M_{\mathcal{P}}(M)$ runs forever; otherwise $M_{\mathcal{P}}(M)$ halts in finite time. We wish to show that there exists some $\hat{\mathcal{P}}$ satisfying EM1 – EM4. Define $\hat{\mathcal{P}}(\chi)$ to be true iff $M_{\mathcal{P}}(\langle\langle\chi\rangle\rangle)$ does not halt in $|\chi|$ steps or less. If χ is infinite, then $\hat{\mathcal{P}}(\chi)$ is true iff $M_{\mathcal{P}}(\langle\langle\chi\rangle\rangle)$ never halts.

$\hat{\mathcal{P}}$ satisfies EM2 because if $M_{\mathcal{P}}(\langle\!\langle\chi\rangle\!\rangle)$ does halt in $|\chi|$ steps or less, then it will also halt in j steps or less whenever $j \geq |\chi|$. $\hat{\mathcal{P}}$ satisfies EM3 because if $M_{\mathcal{P}}(\langle\!\langle\chi\rangle\!\rangle)$ ever halts, it will halt after some finite number of steps. $\hat{\mathcal{P}}$ satisfies EM4 because whenever χ is of finite length, $M_{\mathcal{P}}(\langle\!\langle\chi\rangle\!\rangle)$ can be simulated for $|\chi|$ steps in finite time. Finally, $\hat{\mathcal{P}}$ satisfies EM1 because all and only those PM's $\langle\!\langle\chi\rangle\!\rangle$ that do not satisfy $\mathcal{P}$ cause $M_{\mathcal{P}}$ to halt in time $|\chi|$ for some (sufficiently long) χ. □

Since the coRE properties are a proper superset of the recursively decidable properties [Pap95, p. 63], every statically enforceable policy is trivially enforceable by an EM—the static analysis would be performed by the EM immediately after the PM exhibits its first event (i.e., immediately after the program is loaded). Statically enforceable policies are guaranteed to be computable in a finite period of time, so the EM will always be able to perform this check in finite time and terminate the untrusted PM if the check fails.

Even though the power of the EM approach is strictly greater than that of the static approach, this does not mean that the EM approach is to be universally preferred over the static approach. Depending on the policy to be enforced, either approach might be preferable to the other for engineering reasons. For example, static enforcement mechanisms predict policy violations before a program is run and therefore do not slow the program, whereas EM's usually slow execution due to their added runtime checks. Also, an EM might signal security policy violations arbitrarily late into an execution and only on some executions, whereas a static analysis reveals prior to execution whether that program could violate the policy. Thus, recovering from policy violations discovered by an EM can be more difficult than recovering from those discovered by a static analysis. In particular, an EM might need to roll back a partially completed computation, whereas a static

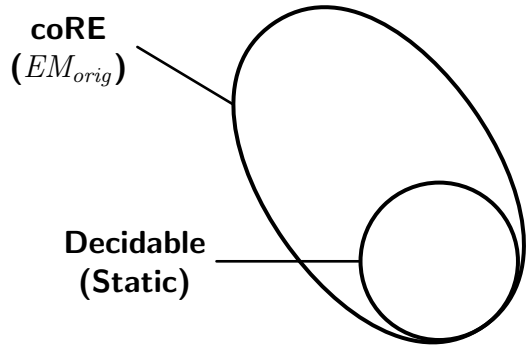


Figure 2.1: The relationship of the static to the coRE policies.

analysis always discovers the violation before computation begins. Alternatively, some policies, though enforceable by a static analysis, are simpler to enforce with an EM, reducing the risk of implementation errors in the enforcement mechanism. Moreover, the comparison of static enforcement to EM-enforcement given in Theorem 2.2 assumes that both are being given the same information (because in our model an EM can examine the untrusted program via event e_M). If a static analysis is provided one representation of the program (e.g., source code) and an EM is provided another in which some of the information has been erased (e.g., object code), then each might well be able to enforce policies that the other cannot.

Theorem 2.2 also suggests that there are policies enforceable by EM's that are not statically enforceable, since $\Pi_0 \subset \Pi_1$. Policy $\mathcal{P}_{boot}$ given in §2.2.2 is an example. More generally, assuring that a PM will never exhibit some prohibited event is equivalent to solving the Halting Problem, which is known to be undecidable and therefore is not statically enforceable. EM's enforce such "undecidable" security policies by waiting until a prohibited event is about to occur, and then signaling the violation.

The relationship of the static policies to class EM_{orig} is depicted in Figure 2.1.

2.3.3 Program-rewriting

An alternative to static analysis or execution monitoring is program-rewriting. A program-rewriter modifies, in finite time, untrusted programs prior to their execution. Use of program-rewriting for enforcing security policies dates back at least to 1969 [DG71]. More recently, we find program-rewriting being employed in software-based fault-isolation (SFI) [WLAG93, Sma97, ABEL05] as a way of implementing memory-safety policies; in Naccio [ET99], SASI [ES00], and Polymer [BLW05] for enforcing security policies in Java; and in Mobile [HMS06a], presented in Chapters 3 and 4, for enforcing security policies in the .NET Framework. Program-rewriters can also be seen as a generalization of execution monitoring, since they can be used to implement an EM as an In-lined Reference Monitor (IRM) whereby an EM is embedded into the untrusted program [ES99]. The approach is appealing, powerful, and quite practical, so understanding what policies it can enforce is a worthwhile goal.

Implicit in program-rewriting is some notion of program equivalence that constrains the program transformations a program-rewriter performs. Executions of the program that results from program-rewriting must have some correspondence to executions of the original. We specify this correspondence in terms of an equivalence relation $\approx$ over PM's. Since there are many notions of PM-equivalence that might be suitable, we leave the definition open, defining relation $\approx$ in terms of an unspecified equivalence relation $\approx_\chi$ on executions:

$$M_1 \approx M_2 =_{\text{def}} \left(\forall \sigma : \sigma \in \Gamma^\omega : \chi_{M_1(\sigma)} \approx_\chi \chi_{M_2(\sigma)}\right) \qquad \text{(PGEQ)}$$

Given a PM-equivalence relation, policies enforceable by program-rewriting are characterized as follows:

Definition 2.2. Let a PM-equivalence relation $\approx$ be given. A policy $\mathcal{P}$ is *$RW_{\approx}$-enforceable* if there exists a total, computable *rewriter function* $R : PM \to PM$ such that for all PM's M,

$$\mathcal{P}(R(M)) \tag{RW1}$$

$$\mathcal{P}(M) \Longrightarrow M \approx R(M) \tag{RW2}$$

Thus, for a security policy $\mathcal{P}$ to be considered $\text{RW}_{\approx}$-enforceable, there must exist a way to transform a PM so that the result is guaranteed to satisfy $\mathcal{P}$ (RW1) and if the original PM already satisfied $\mathcal{P}$, then the transformed PM is equivalent (but not necessarily identical) to the old (RW2). It is important to note that RW1 precludes a program-rewriter from producing as output any PM that does not satisfy the policy. As we shall see in §2.4, this requirement leads to a subtle but important distinction between the class of $\text{RW}_{\approx}$-enforceable policies and EM_{orig}.

Equivalence relation $\approx_\chi$ in PM-equivalence is defined independently of any security policy, but the choice of any particular $\approx_\chi$ places an implicit constraint on which detectors can be considered in any of our analyses of policies enforceable by detectors. In particular, it is sensible to consider only those detectors that are consistent with relation $\approx_\chi$ in the following sense:

Definition 2.3. Detector $\hat{\mathcal{P}}$ is *consistent* with equivalence relation $\approx_\chi$ if it satisfies

$$(\forall \chi_1, \chi_2 : \chi_1, \chi_2 \in E^\omega : \quad \chi_1 \approx_\chi \chi_2 \quad \Longrightarrow \quad \hat{\mathcal{P}}(\chi_1) \equiv \hat{\mathcal{P}}(\chi_2))$$

Detectors consistent with $\approx_\chi$ never classify one execution as acceptable and another as unacceptable when the two are equivalent according to $\approx_\chi$. Program-rewriters presume equivalent executions are interchangeable, which obviously isn't the case if one execution is acceptable and the other is not. Thus, detectors that

are not consistent with $\approx_\chi$ are not compatible with the model. In an analysis of any particular enforcement mechanism involving detectors, $\approx_\chi$ should be defined in such a way that all detectors supported by the mechanism are consistent with $\approx_\chi$, and are therefore covered by the analysis.

The class of $\text{RW}_\approx$-enforceable policies includes virtually all statically enforceable policies.[8] This is because given a statically enforceable policy $\mathcal{P}$, a rewriter function exists that can decide $\mathcal{P}$ directly—that rewriter function returns unchanged any PM that satisfies the policy and returns some safe PM (such as a PM that outputs an error message and terminates) in place of any PM that does not satisfy the policy. This is shown formally below.

Theorem 2.3. *Every satisfiable, statically enforceable policy is $RW_\approx$-enforceable.*

Proof. Let a policy $\mathcal{P}$ be given that is both satisfiable and statically enforceable. Since $\mathcal{P}$ is satisfiable, there exists a program M_1 such that $\mathcal{P}(M_1)$ holds. Define a total function $R : TM \to TM$ by

$$R(M) =_{\text{def}} \begin{cases} M & \text{if } \mathcal{P}(M) \text{ holds} \\ M_1 & \text{if } \neg\mathcal{P}(M) \text{ holds} \end{cases}.$$

R is total because it assigns a TM to every M, and it is computable because $\mathcal{P}$ is statically enforceable and therefore, by Theorem 2.1, recursively decidable. R satisfies RW1 because its range is restricted to programs that satisfy $\mathcal{P}$. Finally, R satisfies RW2 because whenever $\mathcal{P}(M)$ holds, $R(M) = M$. Thus $M \approx R(M)$ holds because $M \approx M$ holds by the reflexivity of equivalence relations. We conclude that $\mathcal{P}$ is $\text{RW}_\approx$-enforceable. □

[8] The one statically enforceable policy not included is the policy that causes all PM's to be rejected, because there would be no PM for R to return.

Theorems 2.2 and 2.3 together imply that the intersection of class EM_{orig} with the $\text{RW}_{\approx}$-enforceable policies includes all satisfiable, statically enforceable policies.

The policies in EM_{orig} that are $\text{RW}_{\approx}$-enforceable policies also include policies that are not statically enforceable, but only for certain notions of PM-equivalence. Program-rewriting is only an interesting method of enforcing security policies when PM-equivalence is a relation that cannot be decided directly. For example, if PM-equivalence is defined syntactically (i.e., two PM's are equivalent if and only if they are structurally identical) then any modification to the untrusted PM produces an inequivalent PM, so RW2 cannot hold. The following theorem shows that if PM-equivalence is a recursively decidable relation, then every $\text{RW}_{\approx}$-enforceable policy that is induced by some detector is statically enforceable. Hence, there is no need to use program-rewriting if PM-equivalence is so restrictive.

Theorem 2.4. *Assume that PM-equivalence relation $\approx$ is recursively decidable, and let $\hat{\mathcal{P}}$ be a detector consistent with $\approx_\chi$. If the policy $\mathcal{P}$ induced by $\hat{\mathcal{P}}$ is $RW_{\approx}$-enforceable then $\mathcal{P}$ is statically enforceable.*

Proof. Exhibit a finite procedure for deciding $\mathcal{P}$, thereby establishing that $\mathcal{P}$ is statically enforceable by Theorem 2.1.

Given M an arbitrary PM, $\mathcal{P}(M)$ can be decided as follows. Start by computing $R(M)$, where R is the program-rewriter given by the $\text{RW}_{\approx}$-enforceability of $\mathcal{P}$. Next, determine if $M \approx R(M)$ which is possible because $\approx$ is recursively decidable, by assumption. If $M \not\approx R(M)$ then RW2 implies that $\neg\mathcal{P}(M)$ holds. Otherwise, if $M \approx R(M)$ then $\mathcal{P}(M)$ holds by the following line of reasoning:

$$\mathcal{P}(R(M)) \qquad \text{(RW1)}$$

$$(\forall \chi : \chi \in X_{R(M)} : \hat{\mathcal{P}}(\chi)) \qquad (\hat{\mathcal{P}} \text{ induces } \mathcal{P}) \qquad (1)$$

$$(\forall\sigma : \sigma \in \Gamma^{\omega} : \chi_{M(\sigma)} \approx_{\chi} \chi_{R(M)(\sigma)}) \qquad \text{(because } M \approx R(M)\text{)}$$

$$(\forall\sigma : \sigma \in \Gamma^{\omega} : \hat{\mathcal{P}}(\chi_{M(\sigma)}) \equiv \hat{\mathcal{P}}(\chi_{R(M)(\sigma)})) \qquad \text{(consistency)} \qquad (2)$$

$$(\forall\sigma : \sigma \in \Gamma^{\omega} : \hat{\mathcal{P}}(\chi_{M(\sigma)})) \qquad \text{(by 1 and 2)} \qquad (3)$$

$$\mathcal{P}(M) \qquad \text{(by 3)}$$

Thus, $\mathcal{P}(M)$ has been decided in finite time and we conclude by Theorem 2.1 that $\mathcal{P}$ is statically enforceable. □

In real program-rewriting enforcement mechanisms, program equivalence is usually defined in terms of execution. For instance, two programs are defined to be *behaviorally equivalent* if and only if, for every input, both programs produce the same output; in a Turing Machine framework, two TM's are defined to be *language-equivalent* if and only if they accept the same language. Both notions of equivalence are known to be Π_2-hard, and other such behavioral notions of equivalence tend to be equally or more difficult. Therefore we assume PM-equivalence is not recursively decidable and not coRE in order for the analysis that follows to have relevance in real program-rewriting implementations.

If PM-equivalence is not recursively decidable, then there exist policies that are $\text{RW}_{\approx}$-enforceable but not statically enforceable. $\mathcal{P}_{boot}$ of §2.2.2, is an example. A rewriting function can enforce $\mathcal{P}_{boot}$ by taking a PM M as input and returning a new PM M' that is exactly like M except just before every computational step of M, M' simulates M for one step into the future on every possible input symbol to see if M will exhibit any prohibited event $\{e_i | 0 \leq i < 512\}$. If any prohibited event is exhibited, then M' is terminated immediately; otherwise the next computational step is performed.

If PM-equivalence is not coRE, then program-rewriting can be used to enforce policies that are not coRE, and therefore not enforceable by any EM.

Theorem 2.5. *There exist non-coRE PM-equivalence relations $\approx$ and policies that are $RW_{\approx}$-enforceable but not coRE.*

Proof. Define $\approx_{\chi}^{TM}$ by $\chi_1 \approx_{\chi}^{TM} \chi_2 \iff \big((e_{end} \in \chi_1) \Leftrightarrow (e_{end} \in \chi_2)\big)$ and define $\approx^{TM}$ according to definition PGEQ. That is, two PM's M_1 and M_2 are equivalent iff they both halt on the same set of input strings. Relation $\approx^{TM}$ defines language-equivalence for Turing Machines, which is known to be Π_2-complete and therefore not coRE. Define M_U to be the universal PM that accepts as input an encoding of an arbitrary PM M and a string σ, whereupon M_U simulates $M(\sigma)$, halting if $M(\sigma)$ halts and looping otherwise. Define policy $\mathcal{P}_U(M) =_{\text{def}} (M \approx^{TM} M_U)$. Observe that policy $\mathcal{P}_U$ is $\text{RW}_{\approx^{TM}}$-enforceable because we can define $R(M) =_{\text{def}} M_U$. Rewriting function R satisfies RW1 because $\mathcal{P}_U(M_U)$ holds, and it satisfies RW2 because if $\mathcal{P}_U(M)$ holds then $M \approx^{TM} M_U$ by construction. We next prove that $\mathcal{P}_U$ is Π_2-complete, and therefore not coRE.

A TM with an oracle that computes the $\approx^{TM}$ relation can trivially compute $\mathcal{P}_U$. We now prove the reverse reduction: A TM with an oracle that computes $\mathcal{P}_U$ can compute the $\approx^{TM}$ relation. Define O_U to be an oracle that, when queried with M, returns true if $\mathcal{P}_U(M)$ holds and false otherwise. Define TM M_{EQ} by $M_{EQ}(M_1, M_2) =_{\text{def}} O_U(M_3)$ where M_3 is defined by

$$M_3(M, \sigma) =_{\text{def}} \begin{cases} M_1(\sigma) & \text{if } M = M_2 \\ M(\sigma) & \text{otherwise} \end{cases}$$

That is, M_{EQ} treats its input tape as an encoding of a pair of PM's M_1 and M_2, and builds a new PM M_3 that is exactly like the universal PM M_U except that

it simulates M_1 when it receives M_2 as input instead of simulating M_2. PM M_{EQ} then queries oracle O_U with M_3. Observe that $M_{EQ} \approx^{TM} M_U$ iff $M_1 \approx^{TM} M_2$. Thus, $M_{EQ}(M_1, M_2)$ holds iff $M_1 \approx^{TM} M_2$. □

The proof of Theorem 2.5 gives a simple but practically uninteresting example of an $\text{RW}_{\approx}$-enforceable policy that is not coRE. Examples of non-coRE, $\text{RW}_{\approx}$-enforceable policies having practical significance do exist. Here is one.

> **The Secret File Policy:** Consider a file system that stores a file whose existence should be kept secret from untrusted programs. Suppose untrusted programs have an operation for retrieving a listing of the directory that contains the secret file. System administrators wish to enforce a policy that prevents the existence of the secret file from being leaked to untrusted programs. So, an untrusted PM satisfies the "secret file policy" if and only if the behavior of the PM is identical to what its behavior would be if the secret file were not stored in the directory.

The policy in this example is not in coRE because deciding whether an arbitrary PM has equivalent behavior on two arbitrary inputs is as hard as deciding whether two arbitrary PM's are equivalent on all inputs. And recall that PM-equivalence ($\approx$) is not coRE. Thus an EM cannot enforce this policy.[9] However, this policy can be enforced by program-rewriting, because a rewriting function never needs to explicitly decide if the policy has been violated in order to enforce the policy. In particular, a rewriter function can make modifications that do not change the

[9] Moreover, an EM cannot enforce this policy by parallel simulation of the untrusted PM on two different inputs, one that includes the secret file and one that does not. This is because an EM must detect policy violations in finite time on each computational step of the untrusted program, but executions can be equivalent even if they are not equivalent step-for-step. Thus, a parallel simulation might require the EM to pause for an unlimited length of time on some step.

behavior of programs that satisfy the policy, but do make safe those programs that don't satisfy the policy. For the example above, program-rewriting could change the untrusted program so that any attempt to retrieve the contents listing of the directory containing the secret file yields an abbreviated listing that excludes the secret file. If the original program would have ignored the existence of the file, then its behavior is unchanged. But if it would have reacted to the secret file, then it no longer does.

The power of program-rewriters is not limitless, however; there exist policies that no program-rewriter can enforce. One example of such a policy is given in the proof of the following theorem.

Theorem 2.6. *There exist non-coRE PM-equivalence relations $\approx$ and policies that are not RW$_{\approx}$-enforceable.*

Proof. Define policy $\mathcal{P}_{NE}(M) =_{\text{def}} \big(\exists\sigma : \sigma \in \Gamma^{\omega} : M(\sigma) \text{ halts}\big)$ and define $\approx^{TM}$ as in the proof of Theorem 2.5 (i.e. $\approx^{TM}$ is defined to be language-equivalence for TM's). Observe that policy $\mathcal{P}_{NE}$ is RE but not recursively decidable. We show that if $\mathcal{P}_{NE}$ were RW$_{\approx^{TM}}$-enforceable, then it would be recursively decidable—a contradiction.

Expecting a contradiction, assume that $\mathcal{P}_{NE}$ is RW$_{\approx^{TM}}$-enforceable. Then there exists a rewriting function R satisfying RW1 and RW2. Use R to decide $\mathcal{P}_{NE}$ in the following way: Let an arbitrary PM M be given. To decide if $\mathcal{P}_{NE}(M)$ holds, construct a new PM M' that treats its input tape as a positive integer i followed by a string σ. (For example, if the input alphabet $\Gamma = \{0, 1\}$, then $i\sigma$ might be encoded as $1^i 0\sigma$. Thus, every possible input string represents some valid encoding of an integer-string pair, and every integer-string pair has some valid encoding.) Upon receiving $i\sigma$ as input, PM M' simulates $M(\sigma)$ for i steps and halts if $M(\sigma)$

halts in that time; otherwise M' loops. Next, compute $R(M')$. Since R satisfies RW1, there exists a string on which $R(M')$ halts. Simulate $R(M')$ on each possible input string for larger and larger numbers of steps until such a string $j\sigma'$ is found. Next, simulate $M(\sigma')$ for j steps. We claim that $\mathcal{P}_{NE}(M)$ holds iff $M(\sigma')$ halts in j steps. If $M(\sigma')$ halts in j steps, then obviously $\mathcal{P}_{NE}(M)$ holds. If $M(\sigma')$ does not halt in j steps, then $M'(j\sigma')$ loops by construction. Since $M'(j\sigma')$ loops but $R(M')(j\sigma')$ halts, it follows that $R(M') \not\approx^{TM} M'$. By RW2, $\mathcal{P}_{NE}(M)$ therefore does not hold.

Thus, we have used R to decide $\mathcal{P}_{NE}$ for arbitrary M, contradicting the fact that $\mathcal{P}_{NE}$ is not recursively decidable. We conclude that no such R exists, and therefore $\mathcal{P}_{NE}$ is not $\text{RW}_{\approx^{TM}}$-enforceable. □

The ability to enforce policies without explicitly deciding them makes the $\text{RW}_{\approx}$-enforceable policies extremely interesting. A characterization of this class in terms of known classes from computational complexity theory would be a useful result, but might not exist. The following negative result shows that, unlike static enforcement and EM_{orig}, no class of the arithmetic hierarchy is equivalent to the class of $\text{RW}_{\approx}$-enforceable policies.

Theorem 2.7. *There exist non-coRE PM-equivalence relations $\approx$ such that the class of $\text{RW}_{\approx}$-enforceable policies is not equivalent to any class of the arithmetic hierarchy.*

Proof. The proof of Theorem 2.5 showed that the Π_2-hard policy $\mathcal{P}_U$ is $\text{RW}_{\approx^{TM}}$-enforceable. Theorem 2.6 showed that the RE policy $\mathcal{P}_{NE}$ is not $\text{RW}_{\approx^{TM}}$-enforceable. Since Π_2 is a superclass of RE, there is no class of the arithmetic hierarchy that includes $\mathcal{P}_U$ but not $\mathcal{P}_{NE}$. We conclude that the class of $\text{RW}_{\approx^{TM}}$-enforceable policies is not equivalent to any class of the arithmetic hierarchy. □

2.4 Execution Monitors as Program-rewriters

Since EM's can be implemented by program-rewriters as in-lined reference monitors [ES00], one might expect EM_{orig} to be a subclass of the $RW_{\approx}$-enforceable policies. However, in this section we show that there are policies in EM_{orig} that are not $RW_{\approx}$-enforceable. In section §2.4.1 we identify some of these policies and argue that they cannot actually be enforced by EM's either. Thus, EM_{orig} is a superclass of the class of policies that EM's can enforce. In §2.4.2 we then show how the definition of $RW_{\approx}$-enforceable policies presented in §2.3.3 can be leveraged to obtain a precise characterization of the EM-enforceable policies.

2.4.1 EM-enforceable Policies

When an EM detects an impending policy violation, it must intervene and prevent that violation. Such an intervention could be modeled as an infinite series of events that gets appended to a PM's execution in lieu of whatever suffix the PM would otherwise have exhibited. Without assuming that any particular set of interventions are available to an EM, let I be the set of possible interventions. Then the policy $\mathcal{P}_I$, that disallows all those interventions, is not enforceable by an EM. If an untrusted program attempts to exhibit some event sequence in I, an EM can only intervene by exhibiting some other event sequence in I, which would in itself violate policy $\mathcal{P}_I$.[10] Nevertheless, $\mathcal{P}_I$ is a member of EM_{orig} as long as I satisfies EM1 – EM4.

[10] In the extreme case that EM's are assumed to be arbitrarily powerful in their interventions, this argument proves only that EM's cannot enforce the unsatisfiable policy. (If $I = E^{\omega}$, then $\mathcal{P}_I$ is the policy that rejects all PM's.) A failure to enforce the unsatisfiable policy might be an uninteresting limitation, but even in this extreme case, EM's have another significant limitation, to be discussed shortly.

For example, [Sch00] presumes that an EM intervenes only by terminating the PM. Define e_{end} to be an event that corresponds to termination of the PM. The policy $\mathcal{P}_{\{e_{end}\}}$, which demands that no PM may terminate, is not enforceable by such an EM even though it satisfies the definition of EM_{orig}. In addition, more complex policies involving e_{end}, such as the policy that demands that every PM must exhibit event e_1 before it terminates (i.e. before it exhibits event e_{end}) are also unenforceable by such an EM, even though they too are members of EM_{orig}. The power of an EM is thus limited by the set of interventions available to it, in addition to the limitations described by the definition of EM_{orig}.

The power of an EM is also limited by its ability to intervene at an appropriate time in response to a policy violation. To illustrate, consider a trusted service that allocates a resource to untrusted programs. When an untrusted program uses the resource, it exhibits event e_{use}. Once the service has allocated the resource to an untrusted program, the service cannot prevent the untrusted program from using the resource. That is, the service cannot revoke access to the resource or halt the untrusted program to prevent subsequent usage. However, untrusted programs can assert that they will no longer use the resource by exhibiting event e_{rel}, after which the service can allocate the resource to another untrusted program.

To avoid two untrusted programs having simultaneous access to the resource, we wish to enforce the policy that the assertions denoted by e_{rel} are always valid. That is, we wish to require that, whenever an untrusted program exhibits event e_{rel}, no possible extension of that execution will subsequently exhibit e_{use}. This would, for example, ensure that no untrusted program retains a reference to the resource that it could use after exhibiting e_{rel}. Formally, we define

$$\hat{\mathcal{P}}_{RU1}(\chi) =_{\text{def}} \left(e_{rel} \notin \chi \ \vee \ (\forall \chi' : \chi\chi' \in X_{\langle\langle \chi \rangle\rangle} : e_{use} \notin \chi')\right)$$

and we wish to enforce the policy $\mathcal{P}_{RU}$ induced by $\hat{\mathcal{P}}_{RU1}$. Enforcing this policy would allow the enforcement mechanism to suppress event e_{rel} when the untrusted program might continue to use the resource, and thereby prevent the service from unsafely allocating the resource to another untrusted program.

One would expect that policy $\mathcal{P}_{RU}$ is not EM-enforceable because detector $\hat{\mathcal{P}}_{RU1}$ is undecidable. Determining, for an arbitrary execution of an arbitrary PM, whether there exists some extension of that execution for that PM that exhibits event e_{use}, is equivalent to solving the Halting Problem. However, policy $\mathcal{P}_{RU}$ is a member of EM_{orig} because the definition of EM_{orig} demands only that *there exists* a detector satisfying EM1 – EM4 that induces the policy, and there is another detector that does so:

$$\hat{\mathcal{P}}_{RU2}(\chi) =_{\text{def}} \left(\forall i : i \geq 1 : (e_{rel} \notin \chi[..i] \ \vee \ e_{use} \notin \chi[i+1..])\right)$$

Detector $\hat{\mathcal{P}}_{RU2}$ rejects executions that have an e_{use} subsequent to an e_{rel}. This detector induces the same policy $\mathcal{P}_{RU}$ because any PM that has an execution that violates detector $\hat{\mathcal{P}}_{RU1}$ will also have a (possibly different) execution that violates detector $\hat{\mathcal{P}}_{RU2}$. Inversely, every PM that has only executions that satisfy $\hat{\mathcal{P}}_{RU1}$ will also have only executions that satisfy $\hat{\mathcal{P}}_{RU2}$. Thus $\hat{\mathcal{P}}_{RU1}$ and $\hat{\mathcal{P}}_{RU2}$ cause the same set of PM's to be accepted or rejected even though they are violated by different executions of those PM's that are rejected.

However, if an EM were to use detector $\hat{\mathcal{P}}_{RU2}$ to enforce policy $\mathcal{P}_{RU}$, it would not be able to prevent two untrusted programs from simultaneously using the resource. An EM using detector $\hat{\mathcal{P}}_{RU2}$ would only discover that an execution should be rejected when the untrusted program attempts to exhibit e_{use} after having exhibited e_{rel} in the past. At that point the service might have already allocated the resource to another untrusted program and would not be able to revoke the

resource from either program. Detector $\hat{\mathcal{P}}_{RU2}$ is therefore violated at a time too late to permit the enforcement mechanism to guarantee that all executions satisfy detector $\hat{\mathcal{P}}_{RU1}$. Violations of $\hat{\mathcal{P}}_{RU1}$ are detected by the EM but cannot be corrected.

In conclusion, an EM can "enforce" policy $\mathcal{P}_{RU}$ in a way that honors detector $\hat{\mathcal{P}}_{RU2}$, but not in a way that honors detector $\hat{\mathcal{P}}_{RU1}$. The definition of EM_{orig} is insufficient to distinguish between a policy induced by $\hat{\mathcal{P}}_{RU1}$ and one induced by $\hat{\mathcal{P}}_{RU2}$ because it places no demands upon the set of executions that results from the composite behavior of the EM executing alongside the untrusted program. The result should be a set of executions that all satisfy the original detector, but EM1 – EM4 can be satisfied even when there is no EM implementation that can accomplish this.

The power of an EM derives from the collection of detectors it offers policy-writers. A small collection of detectors might be stretched to "enforce" all coRE policies according to the terms of EM_{orig}, but in doing this, some of those policies will be "enforced" in ways that permit bad events to occur, which could be unacceptable to those wishing to actually prevent those bad events. Proofs that argue that some real enforcement mechanism is capable of enforcing all policies in EM_{orig} are thus misleading. For example, the MaC system was shown to be capable of enforcing all coRE policies [Vis00], but policies like $\mathcal{P}_{RU}$ cannot be enforced by MaC in such a way as to signal the violations specified by $\hat{\mathcal{P}}_{RU1}$ before the violation has already occurred.

In §2.4.2 we show that the intersection of class EM_{orig} with the $\text{RW}_{\approx}$-enforceable policies constitutes a more suitable characterization of the EM-enforceable policies than class EM_{orig} alone. This is because RW1 and RW2 impose constraints upon an EM's ability to intervene. For example, in a setting where EM's can intervene by suppressing events that would otherwise be exhibited by an untrusted PM, one

might model such interventions by an event $e_{supp(e')}$ that is exhibited whenever an EM suppresses event e'. If EM's cannot suppress e_{use} events, one might wish to enforce policy $\mathcal{P}'_{RU}$ defined by

$$\mathcal{P}'_{RU}(M) =_{\text{def}} \mathcal{P}_{RU}(M) \wedge \left(\forall \sigma : \sigma \in \Gamma^{\omega} : (e_{supp(e_{use})} \notin \chi_{M(\sigma)})\right)$$

By RW1, policy $\mathcal{P}'_{RU}$ is only $\text{RW}_{\approx}$-enforceable if there exists a rewriting function that produces PM's that both satisfy policy $\mathcal{P}_{RU}$ and that never suppress any e_{use} events. Such a constraint on allowable interventions is not expressible using axioms EM1 – EM4 alone because those axioms do not regard the new set of executions that results from an EM's intervention upon an untrusted PM.

Characterizing the EM-enforceable policies as the intersection of class EM_{orig} with the $\text{RW}_{\approx}$-enforceable policies therefore allows us to express policies that regard the whole system rather than just the part of the system that does not include the EM. That is, it allows us to reify the EM into the computation and consider policies that regard this new composite computation rather than just the computation defined by the untrusted PM's behavior in isolation. A enforcement mechanism can only be said to "enforce" a policy if it neither allows any untrusted PM to violate the policy, nor itself violates the policy in the course of "enforcing" it.

This approach also allows us to confirm the intuition that if a policy is EM-enforceable, it should also be enforceable by an in-lined reference monitor. That is, it should be possible to take an EM that enforces the policy and compose it with an untrusted program in such a way that this rewriting process satisfies RW1 and RW2. Axioms RW1 and RW2 require that the computation exhibited by the rewritten PM must satisfy the policy. That is, the composite computation consisting of the original PM's behavior modified by the EM's interventions must satisfy the policy to be enforced.

2.4.2 Benevolent Enforcement of EM-enforceable Policies

To account for the additional restrictions upon EM's described in §2.4.1, it will be useful to identify those detectors for which there is some means to enforce the policies they induce without producing executions that violate the detector. We do so as follows:

Definition 2.4. A detector $\hat{\mathcal{P}}$ is *benevolent* if there exists a decision procedure $M_{\hat{\mathcal{P}}}$ for finite executions such that for all PM's M:

$$\neg\big(\forall\chi : \chi \in X_M : \hat{\mathcal{P}}(\chi)\big) \Longrightarrow \big(\forall\chi : \chi \in X_M^- : (\neg\hat{\mathcal{P}}(\chi) \Rightarrow M_{\hat{\mathcal{P}}}(\chi) \text{ rejects})\big) \quad \text{(B1)}$$

$$\big(\forall\chi : \chi \in X_M : \hat{\mathcal{P}}(\chi)\big) \Longrightarrow \big(\forall\chi : \chi \in X_M^- : (M_{\hat{\mathcal{P}}}(\chi) \text{ accepts})\big) \quad \text{(B2)}$$

A detector that satisfies B1 and B2 can be implemented in such a way that bad events are detected before they occur. In particular, B1 stipulates that an EM implementing detector $\hat{\mathcal{P}}$ rejects all unsafe execution prefixes of an unsafe PM but also permits it to reject unsafe executions early (e.g., if it is able to anticipate a future violation). B1 even allows the EM to conservatively reject some good executions, when a PM does not satisfy the policy. But in order to prevent the EM from being too aggressive in signaling violations, B2 prevents any violation from being signaled when the policy is satisfied.

Detector $\hat{\mathcal{P}}_{RU2}$ of §2.4.1 is an example of a benevolent detector. The decision procedure $M_{\hat{\mathcal{P}}_{RU2}}(\chi)$ would simply scan χ and would reject iff e_{use} was seen after e_{rel}. However, detector $\hat{\mathcal{P}}_{RU1}$ of §2.4.1 is an example of a detector that is not benevolent. It is not possible to discover in finite time whether there exists some extension of execution χ that includes event e_{use} (or, conservatively, whether any execution of $\langle\langle\chi\rangle\rangle$ has an e_{use} after an e_{rel}). Therefore no suitable decision procedure $M_{\hat{\mathcal{P}}_{RU1}}$ satisfying B1 and B2 exists.

Benevolent detectors can be implemented so as to prevent all policy violations without hindering policy-satisfying programs. In the next theorem, we prove that if a policy is both coRE and $\text{RW}_{\approx}$-enforceable for some equivalence relation $\approx$ that permits the sorts of program transformations that are typically performed by in-lined reference monitors, then *every* detector that induces that policy (and that is also consistent with $\approx$ and satisfies EM2) is benevolent. That is, no matter which detector might be desired for enforcing such a policy, there is a way to implement that detector so that all policy violations are prevented but all executions of policy-satisfying programs are permitted. Thus, the class of policies that are both coRE and $\text{RW}_{\approx}$-enforceable constitutes a good characterization of the policies that are actually enforceable by an EM. Such policies can be enforced by an EM that is implemented as an in-lined reference monitor, whereas other coRE policies cannot be so implemented (because they are not $\text{RW}_{\approx}$-enforceable).

We prove this result by first defining a suitable equivalence relation $\approx^{IRM}$. We then prove that any detector that is consistent with $\approx^{IRM}_{\chi}$, that satisfies EM2, and that induces a policy that is both $\text{RW}_{\approx^{IRM}}$-enforceable and coRE, is benevolent. Let $\approx^{IRM}_{\chi}$ be an equivalence relation over executions such that

$$\chi_1 \approx^{IRM}_{\chi} \chi_2 \text{ is recursively decidable}^{11} \text{ whenever } \chi_1 \text{ and } \chi_2 \text{ are both finite.} \quad \text{(EQ1)}$$

$$\chi_1 \approx^{IRM}_{\chi} \chi_2 \Longrightarrow (\forall i \exists j : \chi_1[..i] \approx^{IRM}_{\chi} \chi_2[..j]) \quad \text{(EQ2)}$$

and let $\approx^{IRM}$ be the equivalence relation over programs defined by relation $\approx^{IRM}_{\chi}$ using formula PGEQ.

[11]This assumption can be relaxed to say that $\chi_1 \approx^{IRM}_{\chi} \chi_2$ is recursively enumerable (RE) without affecting any of our results. However, since assuming a recursively decidable relation simplifies several of the proofs, and since we cannot think of a program-equivalence relation of practical interest in which execution-equivalence would not be recursively decidable, we make the stronger assumption of decidability for the sake of expository simplicity.

EQ1 states that although deciding whether two PM's are equivalent might be very difficult in general, an IRM can at least determine whether two individual finite-length execution prefixes are equivalent. EQ2 states that equivalent executions have equivalent prefixes where those prefixes might not be equivalent step for step, reflecting the reality that certain program transformations add computation steps. For example, an IRM is obtained by inserting checks into an untrusted program and, therefore, when the augmented program executes a security check, the behavior of the augmented program momentarily deviates from the original program's. However, assuming the check passes, the augmented program will return to a state that can be considered equivalent to whatever state the original program would have reached.

Theorem 2.8. *Let a detector $\hat{\mathcal{P}}$ satisfying EM2 be given, and assume that $\hat{\mathcal{P}}$ is consistent with $\approx_{\chi}^{IRM}$. If the policy $\mathcal{P}$ induced by $\hat{\mathcal{P}}$ is $RW_{\approx^{IRM}}$-enforceable and satisfies EM1 – EM4, then $\hat{\mathcal{P}}$ is benevolent.*

Proof. Define a decision procedure $M_{\hat{\mathcal{P}}}$ for $\hat{\mathcal{P}}$ and show that it satisfies B1 and B2. We define $M_{\hat{\mathcal{P}}}$ as follows: When $M_{\hat{\mathcal{P}}}$ receives a finite execution prefix χ as input, it iterates through each $i \geq 1$ and for each i, determines if $\chi \approx_{\chi}^{IRM} \chi_{R(\langle\langle\chi\rangle\rangle)(\sigma)}[..i]$, where R is the rewriter given by the $\mathrm{RW}_{\approx^{IRM}}$-enforceability of $\mathcal{P}$ and σ is the string of input symbols recorded in the trace tape as being read during χ. Both of these executions are finite, so by EQ1 this can be determined in finite time. If the two executions are equivalent, then $M_{\hat{\mathcal{P}}}$ halts with acceptance. Otherwise $M_{\hat{\mathcal{P}}}$ simulates $M_{\mathcal{P}}(\langle\langle\chi\rangle\rangle)$ for i steps, where $M_{\mathcal{P}}$ is a TM that halts if and only if its input represents a PM that violates $\mathcal{P}$. Such a TM is guaranteed to exist because $\mathcal{P}$ satisfies EM1 – EM4 and is therefore coRE by Theorem 2.2. If $M_{\mathcal{P}}(\langle\langle\chi\rangle\rangle)$ halts

in i steps, then $M_{\hat{\mathcal{P}}}$ halts with rejection. Otherwise $M_{\hat{\mathcal{P}}}$ continues with iteration $i+1$.

First, we prove that $M_{\hat{\mathcal{P}}}$ always halts. Suppose $\neg\mathcal{P}(\langle\!\langle\chi\rangle\!\rangle)$ holds. Then $M_{\mathcal{P}}$ will eventually reach a sufficiently large i that $M_{\mathcal{P}}(\langle\!\langle\chi\rangle\!\rangle)$ will halt, and thus $M_{\hat{\mathcal{P}}}$ will halt. Suppose instead that $\mathcal{P}(\langle\!\langle\chi\rangle\!\rangle)$ holds. Then by RW1, $\langle\!\langle\chi\rangle\!\rangle \approx^{IRM} R(\langle\!\langle\chi\rangle\!\rangle)$. Applying the definition of $\approx^{IRM}$, we see that $\chi_{\langle\!\langle\chi\rangle\!\rangle(\sigma)} \approx^{IRM}_{\chi} \chi_{R(\langle\!\langle\chi\rangle\!\rangle)(\sigma)}$. Recalling that χ is a finite prefix of $\chi_{\langle\!\langle\chi\rangle\!\rangle(\sigma)}$, observe that EQ2 implies that there exists a (sufficiently large) i such that $\chi \approx^{IRM}_{\chi} \chi_{R(\langle\!\langle\chi\rangle\!\rangle)(\sigma)}[..i]$. Thus $M_{\hat{\mathcal{P}}}$ will halt.

Now observe that the only time $M_{\hat{\mathcal{P}}}$ halts with rejection, $\neg\mathcal{P}(\langle\!\langle\chi\rangle\!\rangle)$ holds. Together with the fact that $M_{\hat{\mathcal{P}}}$ always halts, this establishes that $M_{\hat{\mathcal{P}}}$ satisfies B1.

Finally, we prove that if $M_{\hat{\mathcal{P}}}$ halts with acceptance, then $\hat{\mathcal{P}}(\chi)$ holds. If $M_{\hat{\mathcal{P}}}$ halts with acceptance, then $\chi \approx^{IRM}_{\chi} \chi_{R(\langle\!\langle\chi\rangle\!\rangle)(\sigma)}[..i]$ for some $i \geq 1$. By RW1, $\mathcal{P}(R(\langle\!\langle\chi\rangle\!\rangle))$ holds. Hence $\hat{\mathcal{P}}(\chi_{R(\langle\!\langle\chi\rangle\!\rangle)(\sigma)})$ holds because $\mathcal{P}(R(M)) \equiv (\forall\chi' : \chi' \in X_{R(\langle\!\langle\chi\rangle\!\rangle)} : \hat{\mathcal{P}}(\chi'))$ by assumption, and therefore $\hat{\mathcal{P}}(\chi_{R(\langle\!\langle\chi\rangle\!\rangle)(\sigma)}[..i])$ holds by EM2. Since $\hat{\mathcal{P}}$ is consistent with $\approx^{IRM}_{\chi}$ by assumption, we conclude that $\hat{\mathcal{P}}(\chi)$ holds. This proves that $M_{\hat{\mathcal{P}}}$ satisfies B2. □

The existence of policies in EM_{orig} that are not $\text{RW}_{\approx}$-enforceable can now be shown by demonstrating that there exist coRE policies with detectors that satisfy EM2 but that are not benevolent. By Theorem 2.8, no such policy can be both coRE and $\text{RW}_{\approx^{IRM}}$-enforceable.

Theorem 2.9. *There exist detectors $\hat{\mathcal{P}}$ and equivalence relations $\approx^{IRM}_{\chi}$ such that $\hat{\mathcal{P}}$ is consistent with $\approx^{IRM}_{\chi}$, $\hat{\mathcal{P}}$ satisfies EM2 and EM3, the policy $\mathcal{P}$ induced by $\hat{\mathcal{P}}$ satisfies EM1 – EM4, and yet $\hat{\mathcal{P}}$ is not benevolent.*

Proof. Define $\mathcal{P}_{\{e_{end}\}}$ as in §2.4.1 and define $\hat{\mathcal{P}}_{NT}(\chi) =_{\text{def}} \mathcal{P}_{\{e_{end}\}}(\langle\!\langle \chi \rangle\!\rangle)$. That is, an execution satisfies $\hat{\mathcal{P}}_{NT}$ if and only if it comes from a program that never halts on any input. Define $\approx_{\chi}^{IRM}$ to be the identity relation over executions, and observe that $\hat{\mathcal{P}}_{NT}$ is consistent with $\approx_{\chi}^{IRM}$. Detector $\hat{\mathcal{P}}_{NT}$ satisfies EM2 because for every program M, either all prefixes of all of that program's executions satisfy $\hat{\mathcal{P}}_{NT}$ or none of them do. $\hat{\mathcal{P}}_{NT}$ satisfies EM3 because if an execution falsifies $\hat{\mathcal{P}}_{NT}$, then every finite prefix of that execution falsifies it as well.

Define $\mathcal{P}_{NT}$ to be the policy induced by $\hat{\mathcal{P}}_{NT}$. Observe that $\mathcal{P}_{NT} \equiv \mathcal{P}_{\{e_{end}\}}$ by the following line of reasoning:

$$
\begin{aligned}
\mathcal{P}_{NT}(M) &\equiv (\forall \chi : \chi \in X_M : \hat{\mathcal{P}}_{NT}(\chi)) \\
&\equiv (\forall \chi : \chi \in X_M : \mathcal{P}_{\{e_{end}\}}(\langle\!\langle \chi \rangle\!\rangle)) && \text{(by def of } \hat{\mathcal{P}}_{NT} \text{ above)} \\
&\equiv (\forall \chi : \chi \in X_M : \mathcal{P}_{\{e_{end}\}}(M)) && \text{(because } \chi \in X_M\text{)} \\
&\equiv \mathcal{P}_{\{e_{end}\}}(M) && \text{(by def of } \mathcal{P}_{\{e_{end}\}} \text{ in §2.4.1)}
\end{aligned}
$$

By construction, $\mathcal{P}_{\{e_{end}\}}$ satisfies EM1 – EM4; therefore $\mathcal{P}_{NT}$ satisfies EM1 – EM4.

However, $\hat{\mathcal{P}}_{NT}$ is not benevolent. If it were, then the following would be a finite procedure for deciding the halting problem: For arbitrary M, decide if M ever halts on any input by computing $M_{\hat{\mathcal{P}}_{NT}}(\chi_{M(\sigma)}[..1])$, where $M_{\hat{\mathcal{P}}_{NT}}$ is the decision procedure predicted to exist by the benevolence of $\hat{\mathcal{P}}_{NT}$, and σ is any fixed string. Since $\chi_{M(\sigma)}[..1]$ is finite, $M_{\hat{\mathcal{P}}_{NT}}$ is guaranteed to accept or reject it in finite time. If M never halts on any input, then by B2, $M_{\hat{\mathcal{P}}_{NT}}$ will accept. Otherwise if M does halt on some input, then $\neg\hat{\mathcal{P}}_{NT}(\chi_{M(\sigma)}[..1])$ holds and therefore by B1, $M_{\hat{\mathcal{P}}_{NT}}$ will reject. □

To summarize, the relationship of the statically enforceable policies, the coRE policies (class EM_{orig}), and the policies enforceable by program-rewriting (the

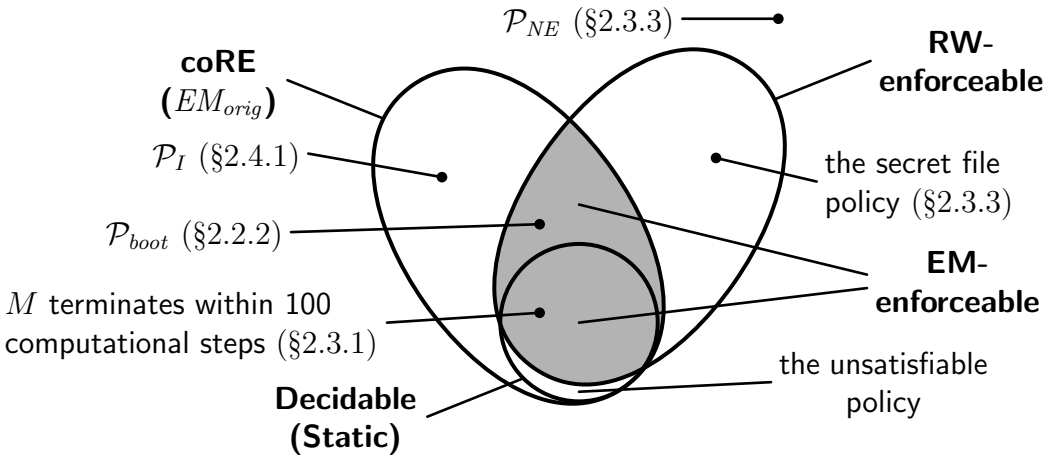


Figure 2.2: Classes of security policies and some policies that lie within them

$RW_{\approx}$-enforceable policies) is shown in Figure 2.2. The statically enforceable policies are a subset of the coRE policies and, with the exception of the unsatisfiable policy, a subset of the $RW_{\approx}$-enforceable policies. The shaded region indicates those policies that are both coRE and $RW_{\approx}$-enforceable. These are the policies that we characterize as EM-enforceable. There exist coRE policies outside this intersection, but all such policies are induced by some non-benevolent detector. Thus, using an EM to "enforce" any of these policies would result in program behavior that might continue to exhibit events that the policy was intended to prohibit. There are also $RW_{\approx}$-enforceable policies outside this intersection. These are policies that cannot be enforced by an EM but that can be enforced by a program-rewriter that does not limit its rewriting to producing in-lined reference monitors.

Figure 2.2 also shows where various specific policies given throughout this chapter lie within the taxonomy of policy classes. The policy "M terminates within 100 computational steps" given in §2.3.1 is an example of a policy that can be enforced by static analysis, execution monitoring, or program-rewriting. Policy $\mathcal{P}_{boot}$,

introduced in §2.2.2, is not enforceable by static analysis, but can be enforced by an EM or by a program-rewriter. The secret file policy described in §2.3.3 is an example of a policy that cannot be enforced by any EM but that can be enforced by a program-rewriter. Finally, policy $\mathcal{P}_I$ is one of the policies given in §2.4.1 that satisfies the definition of EM_{orig} but that cannot be enforced by any real EM in a way that prevents bad events from occurring on the system.

2.5 Related Work

Edit Automata In contrast to program-rewriters, *edit automata* [LBW05a] modify executions rather than modifying programs. Cast in the framework of this dissertation, an edit automaton can intervene at each computational step by inhibiting any event a PM writes to its trace tape and/or writing additional events to the trace tape.

Like program-rewriters, the behavior of an edit automaton is constrained by an equivalence relation over executions. If a PM would have exhibited an execution that satisfied the detector that the edit automaton was to enforce, then any events suppressed or inserted by the edit automaton must result in a final execution that is equivalent to the one that the PM would have exhibited without those suppressions or insertions. But if the PM would have exhibited an execution that falsified the detector, then the edit automaton must suppress or insert events to produce an execution that satisfies the detector. Thus, similar to program-rewriters, edit automata must preserve the semantics of "good" executions whilst preventing "bad" executions.

Edit automata enforce a class of policies called the *infinite renewal policies*—policies for which every infinite policy-satisfying execution has an infinite num-

ber of finite, policy-satisfying prefixes [Lig06, p. 34]. Infinite renewal policies include some policies that are not EM-enforceable, such as some liveness policies [LBW05b, Lig06]. An edit automaton's power above an EM stems from its ability to pause the program being monitored for an infinite length of time, suppressing all remaining events in an infinite sequence unless some "good" event eventually occurs. Our model assumes that an EM must accept or reject each event in finite time, preventing an EM from duplicating this behavior. The infinite renewal policies are a subset of the RW-enforceable policies because program-rewriters can enforce any such policy by in-lining an edit automata into untrusted code, similar to the typical strategy for implementing IRM's.

Shallow History Automata Fong [Fon04] investigates the power of execution monitors that are limited by the information that they can recall but that have no restrictions on their computational power. For example, *shallow history automata* can recall the set of events exhibited so far but not the exact order or number of events exhibited. These and other recall-limited EM's are modeled as automata bound by constraints EM1 – EM3 as well as by recollection constraints. On each computational step they observe any security-relevant event about to be exhibited and either (i) accept, allowing the event to be exhibited and continuing the execution, or (ii) reject, preventing the event from being exhibited and terminating execution. As they decide whether to accept or reject, their recollection constraints render them unable to distinguish between certain finite execution prefixes previously observed. Thus they are weaker than the security automata defined in [Sch00], which are constrained only by EM1 – EM3.

For each different recollection constraint imposed on one of these automata, the automaton can enforce a different subclass of the class of policies defined by EM1 – EM3. The set of all possible such constraints gives rise to a lattice of these subclasses [Fon04]. At the top of the lattice is the subclass equal to the entire class of policies given by EM1 – EM3, corresponding to the automaton that can distinguish between every pair of execution prefixes. At the bottom of the lattice is the subclass consisting of all policies of the form $\mathcal{P}_B$ for some set B of events as defined in §2.2.2—that is, policies that prohibit any of a set of "bad" events from being exhibited. Enforcing such a policy does not require the automaton to recall any history of past events that it observed.

The model proposed in [Fon04] is incomparable to that presented in this chapter because it places no computational constraints on enforcement mechanisms and assumes that all executions are finite. However, if it could be extended to incorporate computational constraints and infinite executions, then this could be used to assess the power of execution monitors that have incomplete access to the event sequences they monitor, such as execution monitors that ignore some of the text of untrusted programs or that cannot observe all non-deterministic choices made by untrusted programs.

Proof-Carrying Code and Certifying Compilers As mentioned in §1.3, Proof-Carrying Code (PCC) [NL96, Nec97] and Certifying Compilers [NL98, MCG99] are emerging technologies for reducing the trusted computing base needed to enforce security policies. They make it easier for code consumers to enforce security policies by requiring code producers to add proof information to the code that they produce. It is thought easier to write trusted code for verifying proofs

than to write trusted code for constructing proofs. PCC is therefore not a single kind of security enforcement mechanism, but rather a framework for reducing and perhaps relocating the trusted computing base. The reader may wonder what policies can be enforced in a PCC framework—that is, the reader may wonder how these technologies fit into the taxonomy of security policies presented in the previous sections. The model and analyses presented in this chapter can be used to explore that question, as we now show.

In a PCC framework, code transmitted to an untrusting code consumer is paired with a proof that the code satisfies whatever policy is being demanded by the code consumer. The code consumer checks that the proof is valid, that the proof concerns the object code, and that the proof establishes the desired policy, all in finite time. Once the code-proof pair has been checked, the code can safely be run without restriction by the code consumer.

The class of policies enforceable by PCC depends on what is the domain of all programs. For the code consumer, the domain of programs is the set of all object code-proof pairs that it might receive. The set of enforceable security policies over this domain are those properties of code-proof pairs that can be decided in finite time. This is the set of recursively decidable ($\Sigma_0 = \Pi_0$) properties of object code-proof pairs, or the statically enforceable policies. (Observe that some policies that are not recursively decidable for code alone are decidable for code-proof pairs. The proof provides extra information that reduces the computational expense of the decision procedure.)

Alternatively, a theorem prover in a PCC framework might consider the domain of programs to be the set of all object programs. The enforceable policies over this domain are those policies such that for all programs that satisfy the policy, there

exists a proof that serves as a witness that the program satisfies the policy. For any proof logic characterizable by some finite axiomization, this is the set of recursively enumerable (Σ_1) properties of that logic. (If an arbitrary program satisfies the policy, this can be discovered in finite time by enumerating all proofs to find a matching one. But if the program doesn't satisfy the policy, the enumeration process will continue indefinitely without finding a suitable proof.)

In practice, code-proof pairs are usually generated together by some automated procedure. For example, certifying compilers [NL98, MCG99] accept a source program and emit not only object code but also a proof that the object code satisfies some policy. If an arbitrary source program satisfies the policy to be enforced, then the certifying compiler must (i) compile it to object code in a way that faithfully preserves its behavior and (ii) generate a matching proof. If the source program doesn't satisfy the policy, then the compiler must either reject the program (which can be thought of as compiling it to the null program) or compile it to some program that does satisfy the policy, possibly by inserting runtime checks that cause the program to change behaviors when some policy violation would otherwise have occurred. These are precisely conditions RW1 and RW2 from the definition of the RW-enforceable policies. Thus, if one considers the domain of programs to be the set of all source code programs received by a certifying compiler or other automated code-proof pair generator, then the set of enforceable policies are the RW-enforceable policies.

2.6 Future Work

The practicality of an enforcement mechanism depends on what resources it consumes. This chapter explored the effects of finitely bounding the space and time

available to various classes of enforcement mechanisms. However, to be considered practical, real enforcement mechanisms must operate in polynomial or even constant space and time. So an obvious extension to the theory presented here is to investigate (i) the set of policies enforceable by program-rewriting when the time and space available to the rewriter is polynomial or constant in the size of the untrusted program and (ii) rewriter functions that produce programs whose size and running time expands by no more than a polynomial or constant in the size and running time of the original untrusted program.

The results of this chapter might also be applied to real enforcement mechanisms. SFI [WLAG93], MiSFIT [Sma97], Naccio [ET99], and SASI/PoET [ES99, ES00] implement program-rewriting but typically assume extremely complex (and mostly unstated) definitions of program equivalence. These equivalence relations would have to be carefully formalized in order to characterize precisely the set of policies that these embodiments of program-rewriting actually enforce. Chapter 4 takes a step in this direction by formally characterizing the class of security policies enforced by the Mobile system in terms of the machinery developed in this chapter.

Finally, the class of RW-enforceable policies outside of the coRE policies remains largely unexplored. To investigate this additional power, program-rewriting mechanisms must be developed. These would need to accept policy specifications that are not limited to the monitoring-style specifications so easily described by a detector. Consequently, there are interesting questions about how to design a suitably powerful yet usable policy specification language for such a system. For example, various meta-level architectures like Aspect Oriented Programming [KLM+97] have been suggested as general frameworks for enforcing a variety of security policies [RVJV99], but it is not clear what class of security policies they can enforce.

2.7 Summary

Our taxonomy of enforceable security policies is depicted in Figure 2.2. We have connected this taxonomy to the arithmetic hierarchy of computational complexity theory by observing that the statically enforceable policies are the recursively decidable properties and that class EM_{orig} is the coRE properties. We also showed that the RW-enforceable policies are not equivalent to any class of the arithmetic hierarchy. The shaded region in Figure 2.2 is argued to be a more accurate characterization of the EM-enforceable policies than EM_{orig}.

Execution monitors implemented as in-lined reference monitors can enforce policies that lie in the intersection of the coRE policies with the RW-enforceable policies. The policies within this intersection are enforceable benevolently—that is, "bad" events are blocked before they occur. But coRE policies that lie outside this intersection might not be benevolently enforceable. In addition, we showed that program-rewriting is an extremely powerful technique in its own right, which can be used to enforce policies beyond those enforceable by execution monitors.

Chapter 3

Mobile: A Type System for Certified Program-rewriting on .NET

The material in this chapter includes previously published [HMS05, HMS06a] joint work with Greg Morrisett and Fred B. Schneider.

3.1 Overview

3.1.1 Certified Program-rewriting

Language-based approaches to computer security [SMH01] have employed two major strategies for enforcing security policies over untrusted code.

- Low-level type systems, such as those used in Java bytecode [LY99], .NET CIL [ECM02], and TAL for x86 [MCG99], can enforce important program invariants such as memory safety and control safety, which dictate that programs must access and transfer control only to certain suitable memory addresses throughout their executions. Proof-Carrying Code (PCC) [NL98] generalizes the type-safety approach by providing an explicit proof of safety in first-order logic.

- Execution Monitoring technologies such as Java and .NET stack inspection [Gon] [LY99, II.22.11], SASI/PoET [ES99, ES00], Java-MAC [KVK+04], Java-MOP [CR05], Polymer [BLW05], and Naccio [ET99], 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 [ES00]. For efficiency, execution monitors are often implemented as In-lined Reference Monitors (IRM's) [Sch00], wherein the runtime checks are in-lined into the untrusted program itself to produce self-monitoring code.

Theorems 2.5 and 2.8 showed that the IRM approach—and, more generally, program-rewriting—is capable of enforcing a large class of powerful security policies, including ones that cannot be enforced with purely static type-checking. In addition, IRM's can enforce a flexible range of policies, often allowing the code recipient to choose the security policy after the code is received, whereas static type systems and PCC usually enforce fixed security policies that are encoded into the type system or proof logic itself, and that therefore cannot be changed without changing the type system or certifying compiler.

However, §1.3 argued that despite their power and flexibility, the *rewriters* that automatically embed IRM's into 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 a significant addition to the system's trusted computing base.

In this chapter, 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) [GS01], a substantial fragment of managed .NET CIL that was used to develop generics for .NET [KS01]. 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 self-monitoring program 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. Even when the rewriter is small and therefore comparable in size to the type-checker, type-checking constitutes a useful level of redundancy that provides greater assurance than trusting the rewriter alone. Mobile thus leverages the power of IRM's while using the type-safety approach to keep the trusted computing base small.

3.1.2 Mobile Security Policies

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. The formalisms presented in this chapter can be leveraged to support many possible languages describing traces, such as deterministic finite automata, security automata [AS87, Sch00], or LTL expressions [Eme90]. The implementation of Mobile discussed in Chapter 4 expresses traces using ω-regular expressions.

A Mobile program *satisfies* the security policy if for every complete run of the program, (i) if the run is finite (i.e., the program terminates), the sequence of security-relevant events performed on every object allocated during that run is a member of the set of traces that the security policy has assigned to that object's class; and (ii) if the run is infinite (i.e., the program does not terminate), at each step of the run the sequence of security-relevant events performed so far on each

security-relevant object is a prefix of a member of the set of traces assigned to that object's class.

One example of a security policy of this form is proposed by [DF04a, p. 5], which prescribes a protocol for proper usage of a `WebPageFetcher` class. The protocol requires code to call the class's `Open` method to acquire a web page resource, call the class's `GetPage` method to use the resource, and call the class's `Close` method to release the resource. A Mobile policy that requires programs to open web pages before reading them, allows at most three reads per opened page, and requires programs to close web pages before the program terminates (but allows them to remain open on runs that never terminate), might assign $(\mathtt{O}\,(\mathtt{G} \cup \mathtt{G}^2 \cup \mathtt{G}^3)\,\mathtt{C})^\omega$ as the set of acceptable traces for class `WebPageFetcher` (where `O`, `G`, and `C` denote `Open`, `GetPage`, 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 network sends may occur during the lifetime of the program. In that case, the global object would additionally be identified as a security-relevant object, a `Send` method call performed on any `System.Net.Sockets.Socket` 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 $\epsilon \cup \mathtt{S} \cup \mathtt{S}^2 \cup \cdots \cup \mathtt{S}^{10}$ (where `S` denotes a `Send` event).

3.1.3 Mobile Type-safety

Policies like the ones described above can only be enforced by a mechanism that tracks events at a per-object level, because the number of security-relevant objects (e.g. the number of `WebPageFetcher` objects) is unbounded and determined at runtime. To statically track the security-relevant state of dynamically allocated objects, Mobile employs a flow-sensitive type system based on typestates [DF04b]. Class types of security-relevant objects are parameterized by an abstraction of the security state of the object at each program point. For example, an object that has type `class WebPageFetcher` in the CIL type system might have type `class WebPageFetcher`$\langle$`opened`$\rangle$ in Mobile to indicate that the object is in state `opened` at a given program point. Security-relevant operations in the code that change the security state of the object at runtime cause that object's type to change along control flows that include that operation.

A type system that tracks per-object security state must also track aliasing of security-relevant objects in order to prevent policy violations. For example, consider the following pseudo-code program in which typing inferences are given in braces.

$\{x \rightarrow \mathit{WebPageFetcher}\langle \mathit{opened} \rangle\}$

1 `x := y`

$\{x \rightarrow \mathit{WebPageFetcher}\langle \mathit{opened} \rangle,\ y \rightarrow \mathit{WebPageFetcher}\langle \mathit{opened} \rangle\}$

2 `x.Close()`

$\{x \rightarrow \mathit{WebPageFetcher}\langle \mathit{closed} \rangle,\ y \rightarrow \mathit{WebPageFetcher}\langle ??? \rangle\}$

3 `y.GetPage()`

If the type system cannot infer that x and y are aliases for the same security-relevant object after line 1, then it would incorrectly infer that y is still in the *closed* state after line 2 and would permit the policy-violating operation in line 3 that attempts to retrieve a web page from a closed *WebPageFetcher* object.

To track simple aliasing like that illustrated by the program above, Mobile's type system employs an extra level of indirection based on alias types [SWM00]:

$\{x \rightarrow \mathit{WebPageFetcher}\langle\ell\rangle\}\{\ell \rightarrow \mathit{opened}\}$

1 `x := y`

$\{x \rightarrow \mathit{WebPageFetcher}\langle\ell\rangle,\ y \rightarrow \mathit{WebPageFetcher}\langle\ell\rangle\}\{\ell \rightarrow \mathit{opened}\}$

2 `x.Close()`

$\{x \rightarrow \mathit{WebPageFetcher}\langle\ell\rangle,\ y \rightarrow \mathit{WebPageFetcher}\langle\ell\rangle\}\{\ell \rightarrow \mathit{closed}\}$

3 `y.GetPage()`

Type variable ℓ tracks the security-relevant state of all aliases of x, permitting the type system to infer that line 3 constitutes a policy violation because y refers to an object in state ℓ, and ℓ refers to the *closed* state.

The above scheme suffices to track aliases when security-relevant objects are reachable by a bounded graph rooted at the program variables, but it does not suffice when security-relevant objects might escape to the heap. For example, if the program maintains a linked list of *WebPageFetcher* objects, all of which might be in different security-relevant states, then the type system must have some way to statically infer the security-relevant state of objects retrieved from that list.

To allow code to let security-relevant objects escape to the heap in such a way that the type system can correctly infer each object's security-relevant state, Mobile supports a **pack** operation that pairs a security-relevant object with a

runtime value (e.g., an integer) representing the object's current state, and then encapsulates them into a two-field *package* object. Packages can be aliased arbitrarily, providing well-typed Mobile code a means to safely allow security-relevant objects to escape to the heap or share security-relevant objects between threads. For example, the following pseudo-code stores a security-relevant object x into a linked list by first packing it:

$\{x \rightarrow \mathit{WebPageFetcher}\langle\ell\rangle\}\{\ell \rightarrow \mathit{state32}\}$

1 `p` := `newpackage WebPageFetcher`

2 `p.Pack(x, 32)`

$\{x \rightarrow \mathit{WebPageFetcher}\langle\ell\rangle,\ p \rightarrow \mathit{package\ WebPageFetcher}\}\{\ell \rightarrow \mathit{revoked}\}$

3 `list.item` := `p`

Notice that once packed, object x cannot be accessed again directly because typing variable ℓ has been revoked from the typing context. This is to prevent security-relevant operations on objects that are packed, since such an operation could change the object's security-relevant state without changing the runtime state value stored with it in the package. A package class's 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.

Mobile's **unpack** operation can be used to unpack a package, yielding the original object that was packed along with the runtime value that represents its state. To prevent untracked aliases from being introduced to the typing context, the **unpack** operation is implemented as a destructive read, preventing the package from being unpacked twice before it is re-packed. For example, the following code retrieves a *WebPageFetcher* object from a list:

```
1   (x, n) := list.item.Unpack()
    {x → WebPageFetcher⟨ℓ⟩, n → Rep⟨θ⟩}{ℓ → θ}
2   if n = 32 then
        {x → WebPageFetcher⟨ℓ⟩, n → 32}{ℓ → state32}
3       x.GetPage()
```

Observe that when the package is first unpacked, the type system cannot initially infer the security-relevant state of object x. All that is statically known is that n is an integer representation of whatever state x is in. History type variable θ is introduced to the typing context to denote this unknown state. If integer n is dynamically tested as in line 2, then the typing context can be refined within the branches of the conditional. In the case of the program above, the type system can statically determine that if control reaches line 3 at runtime, then object x must be in state 32. Security-relevant operations can be placed in the branches of such conditionals, providing Mobile programs a means to guard potentially dangerous security-relevant operations with runtime security checks.

To summarize, Mobile allows only limited aliasing of unpacked security-relevant objects—none of their aliases can escape to the heap. Packages, however, are permitted to escape to the heap and to undergo unlimited aliasing. The memory safety and control-flow safety guarantees provided by the CIL type system ensure that Mobile code obeys a package's interface. 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 (or their finalizers must bring them to a policy-adherent state at program termination; see §4.1), whereas 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.

Implementing **pack** and **unpack** as atomic swaps is a somewhat blunt approach, but it is still powerful enough to support useful and effective rewriting strategies. Using the above operations, a naïve rewriter can implement state-based histories by simply representing security-relevant objects as packages. Whenever a security-relevant operation is to be performed, the rewriter would insert code to first unpack the package and test the object's runtime state, then perform the security-relevant operation only if the test succeeds (possibly terminating otherwise), and finally repackage the object with updated state.

This strategy suffices to implement any state-based history but might result in inefficient code if security-relevant operations are frequent. Thus, Mobile's type system also makes it possible to avoid some of these dynamic operations when policy-adherence can be proved statically. For example, a more sophisticated rewriter could in some cases insert code to perform numerous security-relevant operations consecutively without any dynamic checks. Instead of dynamic checks, the rewriter could add typing annotations that prove to the type-checker that the omitted checks are unnecessary for preventing a security violation. Substituting

annotations for dynamic checks in this way is often possible in straight-line code or tight loops that do not leak security-relevant objects to the heap. However, when objects do escape to the heap, the type system is not sufficiently powerful to track them and dynamic checks would usually be necessary in order to prove that a security violation cannot occur. Thus, Mobile's type system is sufficiently expressive that rewriters can avoid some but not all dynamic checks.

3.2 Formal Definition of Mobile

3.2.1 The Abstract Machine

Figure 3.1 gives the Mobile instruction set. Like BIL, Mobile's syntax is written in postfix notation. Postfix notation is suggestive of the behavior of the .NET virtual machine, which employs an evaluation stack. For example, the Mobile program

$$(\textbf{ldarg}\ 0)\ (\textbf{ldc.i4}\ 5)\ \textbf{stfld int32}\ C::f$$

stores a 5 into the f field of the object given by argument 0. This is because term (**ldarg** 0) returns the value of argument 0, term (**ldc.i4** 5) returns the integer value 5, and term $I_0\ I_1$ **stfld int32** $C::f$ stores the value returned by term I_1 into field f of the value returned by term I_0. A CIL program that performs the same function would first execute an instruction to load argument 0 onto the evaluation stack, then execute an instruction that loads the constant 5 onto the evaluation stack, and finally execute an instruction that pops the top two arguments off the evaluation stack, storing the second into a named field of the first. Observe that this procedure is analogous to executing the Mobile program above as straight-line code. It is hence possible to obtain CIL programs from BIL programs via a series of trivial syntactic transformations, as shown in [GS01].

$I ::= \textbf{ldc.i4}\ n$	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

Figure 3.1: The Mobile instruction set

In addition to BIL instructions,[1] Mobile includes

- instruction **evt** e, which performs security-relevant operation e on an object (where e is some unique identifier, such as "e_{open}", that we associate with each security-relevant operation),
- 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 §3.2.2. (Instruction $\boxed{v}$ is a term that has been reduced to value v, and instruction **ret** pops the current stack frame at the end of a method call.)

These abstract instructions model real CIL instructions. For example, if calls to method m are security-relevant operations, the CIL instruction that invokes m on object o is modeled by the Mobile instruction sequence

$$o \textbf{ evt } e_m;\ o \textbf{ callvirt } C{::}m.Sig_m$$

A description of how our implementation models other CIL instructions is given in Chapter 4.

Figure 3.2 provides Mobile's type system. Mobile types consist of void types, integers, classes, and *history abstractions* (the types of runtime state values). The

[1]For simplicity, we omit BIL's value classes and managed pointers from Mobile, but otherwise include all BIL types and instructions.

Types	$\tau ::= \mu \mid C\langle \ell \rangle$
Untracked types	$\mu ::= \mathbf{void} \mid \mathbf{int32} \mid C\langle ? \rangle \mid \mathcal{R}ep_C\langle H \rangle$
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 3.2: The Mobile type system

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

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

$$\frac{\tau_i \preceq \tau'_i \;\; \forall i \in 0..n}{(\tau_0, \ldots, \tau_n) \preceq (\tau'_0, \ldots, \tau'_n)}$$

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

Figure 3.3: Mobile subtyping

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 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* 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 3.3. Observe that the subtyping relation in Figure 3.3 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 declare 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. Note that history maps do not typically track object traces precisely. Instead, they associate with each object identity variable a conservative approximation of the set of traces that might have been exhibited on the object by the time control reaches any given program point. Additionally, history maps do not typically include a mapping for every security-relevant object in memory. Rather, they track only those security-relevant objects that are in scope at a given program point and are not packed. History maps therefore do not record the exact security state of the entire system, but they permit local reasoning about the security state. Since object identity variables uniquely identify each object instance, object history maps can be seen as a spatial conjunction ($\star$) [ORY01] of assertions about the histories of various unpacked objects in the heap.

A complete Mobile program consists of:

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

We also use the notation $\mathit{fields}(C)$ to refer to the number of fields in class C. Method signatures Sig will be described in §3.2.3.

$v ::=$	result
$\quad \mathbf{0}$	void
$\quad i4$	integer
$\quad \ell$	heap pointer
$\quad \mathit{rep}_C(H)$	runtime state value
$o ::=$	heap elements
$\quad obj_C\{f_i = v_i\}^{\overrightarrow{e}}$	object
$\quad pkg(\ell, \mathit{rep}_C(H))$	filled package
$\quad pkg(\cdot)$	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 3.4: The Mobile memory model

3.2.2 Operational Semantics

Unlike [GS01], 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 §3.4 to programs that do not terminate or that enter a bad state.

In Mobile's small-step memory model, presented in Figure 3.4, objects consist not only of an assignment of values to fields but also a trace $\overrightarrow{e}$ that records an exact history of the security-relevant operations performed on the object. Although our model attaches a history trace to each object, we prove in §3.4 that it is unnecessary

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

Figure 3.5: Mobile evaluation contexts

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 3.5 and 3.6, defines how a given store ψ and instruction I steps to a new store ψ' and instruction I', written $\psi, I \rightsquigarrow \psi', I'$. Rules 3.2 – 3.12 model the behavior of instructions in BIL and rules 3.13 – 3.18 model the behavior of the new instructions introduced by Mobile. Rule 3.1 and Figure 3.5 capture the usual semantics of the order of evaluation of instruction arguments by representing a partially evaluated instruction as an evaluation context where evaluation resumes at the first unevaluated argument. The remaining rules then model how evaluation proceeds once an instruction's relevant arguments have been reduced to values. Some intuition behind each of these rules is provided below.

Rule 3.2 steps an **ldc.i4** instruction to an integer constant value. Rule 3.3 evaluates the positive branch of a conditional if the test argument evaluates to

Figure 3.6: Small-step operational semantics for Mobile

$$\frac{\psi, I \rightsquigarrow \psi', I'}{\psi, E[I] \rightsquigarrow \psi', E[I']} \tag{3.1}$$

$$\psi, \textbf{ldc.i4}\ i4 \rightsquigarrow \psi, \boxed{i4} \tag{3.2}$$

$$\frac{\text{if } i4 = 0 \text{ then } j = 3 \text{ else } j = 2}{\psi, \boxed{i4}\ I_2\ I_3\ \textbf{cond} \rightsquigarrow \psi, I_j} \tag{3.3}$$

$$\psi, I_1\ I_2\ \textbf{while} \rightsquigarrow \psi, I_1\ (I_2; (I_1\ I_2\ \textbf{while}))\ \boxed{\textbf{0}}\ \textbf{cond} \tag{3.4}$$

$$\psi, \boxed{v}; I_2 \rightsquigarrow \psi, I_2 \tag{3.5}$$

$$\frac{0 \leq j \leq n}{(h, s(v_0, \ldots, v_n)), \textbf{ldarg}\ j \rightsquigarrow (h, s(v_0, \ldots, v_n)), \boxed{v_j}} \tag{3.6}$$

$$\frac{0 \leq j \leq n}{\begin{array}{l}(h, s(v_0, \ldots, v_n)), \boxed{v}\ \textbf{starg}\ j \rightsquigarrow \\ \quad (h, s(v_0, \ldots, v_{j-1}, v, v_{j+1}, \ldots, v_n)), \boxed{\textbf{0}}\end{array}} \tag{3.7}$$

$$\frac{\ell \notin \mathit{Dom}(h) \qquad n = \mathit{fields}(C)}{\begin{array}{l}(h, s), \boxed{v_1}\ \ldots\ \boxed{v_n}\ \textbf{newobj}\ C(\mu_1, \ldots, \mu_n) \rightsquigarrow \\ \qquad (h[\ell \mapsto \mathit{obj}_C\{f_i = v_i | i \in 1..n\}^{\epsilon}], s), \boxed{\ell}\end{array}} \tag{3.8}$$

$$\frac{\mathit{methodbody}(C::m.Sig) = I}{(h, s), \boxed{v_0}\ \ldots\ \boxed{v_n}\ \textbf{callvirt}\ C::m.Sig \rightsquigarrow (h, s(v_0, \ldots, v_n)), I\ \textbf{ret}} \tag{3.9}$$

$$(h, sa), \boxed{v}\ \textbf{ret} \rightsquigarrow (h, s), \boxed{v} \tag{3.10}$$

$$\frac{h(\ell) = \mathit{obj}_C\{\ldots, f = v, \ldots\}^{\overrightarrow{e}}}{(h, s), \boxed{\ell}\ \textbf{ldfld}\ \mu\ C::f \rightsquigarrow (h, s), \boxed{v}} \tag{3.11}$$

$$\frac{h(\ell) = \mathit{obj}_C\{\ldots, f = v, \ldots\}^{\overrightarrow{e}}}{(h, s), \boxed{\ell}\ \boxed{v'}\ \textbf{stfld}\ \mu\ C::f \rightsquigarrow (h[\ell \mapsto \mathit{obj}_C[f \mapsto v']], s), \boxed{\textbf{0}}} \tag{3.12}$$

Figure 3.6 (Continued)

$$\frac{h(\ell) = obj_C\{\ldots\}^{\overrightarrow{e}}}{(h,s), \boxed{\ell}\ \mathbf{evt}\ e_1 \rightsquigarrow (h[\ell \mapsto obj_C\{\ldots\}^{\overrightarrow{e}\, e_1}], s), \boxed{\mathbf{0}}} \tag{3.13}$$

$$\frac{\ell \notin Dom(h)}{(h,s), \mathbf{newpackage}\ C \rightsquigarrow (h[\ell \mapsto pkg(\cdot)], s), \boxed{\ell}} \tag{3.14}$$

$$\frac{h(\ell) = pkg(\ldots)}{(h,s), \boxed{\ell}\ \boxed{\ell'}\ \boxed{\mathit{rep}_C(H)}\ \mathbf{pack} \rightsquigarrow (h[\ell \mapsto pkg(\ell', \mathit{rep}_C(H))], s), \boxed{\mathbf{0}}} \tag{3.15}$$

$$\frac{h(\ell) = pkg(\ell', \mathit{rep}_C(H)) \qquad 0 \le j \le n}{\begin{array}{l}(h, s(v_0, \ldots, v_n)), \boxed{\ell}\ \mathbf{unpack}\ j \rightsquigarrow \\ \quad \big(h[\ell \mapsto pkg(\cdot)], s(v_0, \ldots, v_{j-1}, \mathit{rep}_C(H), v_{j+1}, \ldots, v_n)\big), \boxed{\ell'}\end{array}} \tag{3.16}$$

$$\frac{\text{if } test_{C,k}(\mathit{rep}_C(H)) = 0 \text{ then } j = 3 \text{ else } j = 2}{\psi, \boxed{\mathit{rep}_C(H)}\ I_2\ I_3\ \mathbf{condst}\ C, k \rightsquigarrow \psi, I_j} \tag{3.17}$$

$$\frac{arity(hc_{C,k}) = n}{\psi, \boxed{v_1}\ \ldots\ \boxed{v_n}\ \mathbf{newhist}\ C, k \rightsquigarrow \psi, \boxed{hc_{C,k}(v_1, \ldots, v_n)}} \tag{3.18}$$

true (non-zero) or evaluates the negative branch, otherwise. Rule 3.4 expands one iteration of a **while** loop into a conditional that, if true, executes the loop body and reiterates, and otherwise yields a void value. Rule 3.5 discards the first instruction in a sequence once it has been reduced to a value so that evaluation can continue with the next instruction in the sequence. Rule 3.6 causes the **ldarg** j instruction to retrieve the value in slot j of the bottom frame of the stack, and Rule 3.7 causes the **starg** j instruction to store a value into slot j of the bottom frame of the stack.

Rule 3.8 is the first rule to touch the heap. It creates a fresh heap pointer ℓ and assigns it a fresh object of class C whose fields have all been initialized to prescribed values. Object ℓ is both added to the heap and returned as the result. Rules 3.11 and 3.12 load and store fields of objects by retrieving a given object ℓ from the heap and either reading the value v of field f or storing a given value v' into it.

Rules 3.9 and 3.10 model instance method calls and returns. (Static method calls and returns are omitted for simplicity.) Rule 3.9 calls method m by retrieving the method body I associated with m and in-lining it into the partially evaluated term. It adds a new frame $(v_0, \ldots, v_n)$ to the bottom of the stack consisting of the arguments with which m is called. Method body I is enclosed in a **ret** instruction so that when evaluation reduces the method body to a value, Rule 3.10 will pop the method's stack frame off the bottom of the stack.

Rule 3.13 models the exhibition of event e_1 on object ℓ by appending event e_1 to the sequence of events in the object trace recorded for ℓ. (Recall that this trace is not explicitly recorded by an implementation of Mobile. It is represented in these

formalisms to permit formal reasoning about the policy adherence of well-typed Mobile programs.)

Rules 3.14 – 3.16 model packages. Rule 3.14 introduces a new package object to the heap and assigns it a fresh heap pointer ℓ. Rule 3.15 fills a package ℓ by assigning it an object ℓ' and a runtime state value $\mathit{rep}_C(H)$. Rule 3.16 unpacks a package ℓ by yielding the object ℓ' and runtime state value $\mathit{rep}_C(H)$ within it. Since Mobile's type system does not include pairs, object ℓ' is returned as the result of the instruction and runtime state value $\mathit{rep}_C(H)$ is returned by assigning it to slot j of the local argument frame. Unpacking a package empties it; unpacking an empty package is an error and causes the virtual machine to enter a stuck state.

Rules 3.17 and 3.18 model runtime state values and introduce new notation that deserves special note. Runtime operations $test_{C,k}$ and $hc_{C,k}$ 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 $test_{C,k}$ and $hc_{C,k}$ can therefore be used to allow Mobile to support different collections of security policies. In §3.2.4 we show that a Mobile system that implements runtime state values as integers can instantiate these runtime operations to support security policies expressed as deterministic finite automata or as resource bounds. 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 chapter, we assume only that a countable collection of state value constructors and tests exists and that this collection adheres to typing constraints 3.42, 3.43, 3.44, and 3.45 presented in §3.2.4.

Rule 3.17 tests runtime state values and differs from Rule 3.3 only in that runtime operation $test_{C,k}$ is consulted to determine if the test succeeds. Rule 3.18 creates a new runtime history value by invoking history constructor operation $hc_{C,k}$.

The operational semantics given in Figure 3.6 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 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. Mobile can be implemented so as to support concurrency and finalizers as we show in Chapter 4, but to simplify the presentation, we leave the analysis of these language features to future work.

3.2.3 Type System

Mobile's type system considers each 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 result of type τ' (if it terminates). Any new typing variables appearing in $\overrightarrow{Fr}'$ and τ' are bound in context Γ'. A method signature (see Figure 3.2) is the type assigned to the term comprising its body.

Figure 3.7: Typing rules for Mobile

$$\frac{}{\Gamma \vdash \textbf{ldc.i4}\ n : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; \textbf{int32})} \tag{3.19}$$

$$\frac{\begin{array}{c}\Gamma \vdash I_1 : (\Psi; \overrightarrow{Fr}) \multimap \exists \Gamma_1.(\Psi_1; \overrightarrow{Fr}_1; \textbf{int32}) \\ \Gamma, \Gamma_1 \vdash I_i : (\Psi_1; \overrightarrow{Fr}_1) \multimap \exists \Gamma'.(\Psi'; \overrightarrow{Fr}'; \tau)\ \forall i \in \{2,3\}\end{array}}{\Gamma \vdash I_1\ I_2\ I_3\ \textbf{cond} : (\Psi; \overrightarrow{Fr}) \multimap \exists \Gamma_1, \Gamma'.(\Psi'; \overrightarrow{Fr}'; \tau)} \tag{3.20}$$

$$\frac{\Gamma, \Gamma' \vdash I_1\ I_2\ \boxed{\textbf{0}}\ \textbf{cond} : (\Psi; \overrightarrow{Fr}) \multimap \exists \Gamma'.(\Psi; \overrightarrow{Fr}; \textbf{void})}{\Gamma, \Gamma' \vdash I_1\ I_2\ \textbf{while} : (\Psi; \overrightarrow{Fr}) \multimap \exists \Gamma'.(\Psi; \overrightarrow{Fr}; \textbf{void})} \tag{3.21}$$

$$\frac{\begin{array}{c}\Gamma \vdash I_1 : (\Psi; \overrightarrow{Fr}) \multimap \exists \Gamma_1.(\Psi_1; \overrightarrow{Fr}_1; \textbf{void}) \\ \Gamma, \Gamma_1 \vdash I_2 : (\Psi_1; \overrightarrow{Fr}_1) \multimap \exists \Gamma_2.(\Psi'; \overrightarrow{Fr}'; \tau)\end{array}}{\Gamma \vdash I_1; I_2 : (\Psi; \overrightarrow{Fr}) \multimap \exists \Gamma_1, \Gamma_2.(\Psi'; \overrightarrow{Fr}'; \tau)} \tag{3.22}$$

$$\frac{\begin{array}{c}\ell \in \mathit{Dom}(\Psi') \qquad \mathit{field}(C, f) = \mu \\ \Gamma \vdash I : (\Psi; \overrightarrow{Fr}) \multimap \exists \Gamma'.(\Psi'; \overrightarrow{Fr}'; C\langle \ell \rangle)\end{array}}{\Gamma \vdash I\ \textbf{ldfld}\ \mu\ C{::}f : (\Psi; \overrightarrow{Fr}) \multimap \exists \Gamma'.(\Psi'; \overrightarrow{Fr}'; \mu)} \tag{3.23}$$

$$\frac{\begin{array}{c}\Gamma \vdash I_1 : (\Psi; \overrightarrow{Fr}) \multimap \exists \Gamma_1.(\Psi_1; \overrightarrow{Fr}_1; C\langle \ell \rangle) \qquad \ell \in \mathit{Dom}(\Psi') \\ \Gamma, \Gamma_1 \vdash I_2 : (\Psi_1; \overrightarrow{Fr}_1) \multimap \exists \Gamma_2.(\Psi'; \overrightarrow{Fr}'; \mu) \quad \mathit{field}(C, f) = \mu\end{array}}{\Gamma \vdash I_1\ I_2\ \textbf{stfld}\ \mu\ C{::}f : (\Psi; \overrightarrow{Fr}) \multimap \exists \Gamma_1, \Gamma_2.(\Psi'; \overrightarrow{Fr}'; \textbf{void})} \tag{3.24}$$

$$\frac{0 \leq j \leq n}{\Gamma \vdash \textbf{ldarg}\ j : (\Psi; \overrightarrow{Fr}(\tau_0, \ldots, \tau_n)) \multimap (\Psi; \overrightarrow{Fr}(\tau_0, \ldots, \tau_n); \tau_j)} \tag{3.25}$$

$$\frac{\Gamma \vdash I : (\Psi; \overrightarrow{Fr}) \multimap \exists \Gamma'.(\Psi'; \overrightarrow{Fr}'(\tau_0, \ldots, \tau_n); \tau) \qquad 0 \leq j \leq n}{\begin{array}{l}\Gamma \vdash I\ \textbf{starg}\ j : (\Psi; \overrightarrow{Fr}) \multimap \\ \qquad \exists \Gamma'.(\Psi'; \overrightarrow{Fr}'(\tau_0, \ldots, \tau_{j-1}, \tau, \tau_{j+1}, \ldots, \tau_n); \textbf{void})\end{array}} \tag{3.26}$$

$$\frac{\begin{array}{c}\Gamma, \Gamma_1, \ldots, \Gamma_{i-1} \vdash I_i : (\Psi_{i-1}; \overrightarrow{Fr}_{i-1}) \multimap \\ \qquad \exists \Gamma_i.(\Psi_i; \overrightarrow{Fr}_i; \mu_i) \quad \forall i \in 1..n \\ n = \mathit{fields}(C) \quad \ell \notin \mathit{Dom}(\Gamma, \Gamma_1, \ldots, \Gamma_n) \quad \epsilon \in \mathit{pre}(\mathit{policy}(C))\end{array}}{\begin{array}{l}\Gamma \vdash I_1\ \ldots\ I_n\ \textbf{newobj}\ C(\mu_1, \ldots, \mu_n) : \\ \quad (\Psi_0; \overrightarrow{Fr}_0) \multimap \exists \Gamma_1, \ldots, \Gamma_n, \ell{:}C.(\Psi_n \star (\ell \mapsto \epsilon); \overrightarrow{Fr}_n; C\langle \ell \rangle)\end{array}} \tag{3.27}$$

Figure 3.7 (Continued)

$$\frac{\begin{array}{c}\Gamma_0,\ldots,\Gamma_j \vdash I_j : (\Psi_j, \overrightarrow{Fr}_j) \multimap \exists \Gamma_{j+1}.(\Psi_{j+1}, \overrightarrow{Fr}_{j+1}, \tau_j)\ \forall j \in 0..n \\ \tau_0 = C\langle \ell \rangle \quad \ell \in Dom(\Psi_{n+1}) \quad C{::}m.Sig \in Dom(methodbody) \\ \Gamma_0,\ldots,\Gamma_n \vdash Sig <: (\Psi_{in}, (\tau_0,\ldots,\tau_n)) \multimap \exists \Gamma_{out}.(\Psi_{out}, Fr_{out}, \tau) \\ \Psi_{n+1} = \Psi_{unused} \star \Psi_{in}\end{array}}{\begin{array}{l}\Gamma_0 \vdash I_0\ \ldots\ I_n\ \textbf{callvirt}\ C{::}m.Sig : \\ \quad (\Psi_0, \overrightarrow{Fr}_0) \multimap \exists \Gamma_1,\ldots,\Gamma_{n+1},\Gamma_{out}.(\Psi_{unused} \star \Psi_{out}, \overrightarrow{Fr}_{n+1}, \tau)\end{array}} \tag{3.28}$$

$$\frac{\begin{array}{c}He \subseteq pre(policy(C)) \\ \Gamma \vdash I : (\Psi; \overrightarrow{Fr}) \multimap \exists \Gamma'.(\Psi' \star (\ell \mapsto H); \overrightarrow{Fr}'; C\langle \ell \rangle)\end{array}}{\Gamma \vdash I\ \textbf{evt}\ e : (\Psi; \overrightarrow{Fr}) \multimap \exists \Gamma'.(\Psi' \star (\ell \mapsto He); \overrightarrow{Fr}'; \textbf{void})} \tag{3.29}$$

$$\frac{\ell \notin Dom(\Gamma)}{\Gamma \vdash \textbf{newpackage}\ C : (\Psi; \overrightarrow{Fr}) \multimap \exists \ell{:}C\langle ? \rangle.(\Psi; \overrightarrow{Fr}; C\langle ? \rangle)} \tag{3.30}$$

$$\frac{\begin{array}{c}H \subseteq H' \subseteq policy(C) \\ \Gamma \vdash I_1 : (\Psi; \overrightarrow{Fr}) \multimap \exists \Gamma_1.(\Psi_1; \overrightarrow{Fr}_1; C\langle ? \rangle) \\ \Gamma, \Gamma_1 \vdash I_2 : (\Psi_1; \overrightarrow{Fr}_1) \multimap \exists \Gamma_2.(\Psi_2; \overrightarrow{Fr}_2; C\langle \ell \rangle) \\ \Gamma, \Gamma_1, \Gamma_2 \vdash I_3 : (\Psi_2; \overrightarrow{Fr}_2) \multimap \exists \Gamma_3.(\Psi' \star (\ell \mapsto H); \overrightarrow{Fr}'; \mathcal{Rep}_C\langle H' \rangle)\end{array}}{\Gamma \vdash I_1\ I_2\ I_3\ \textbf{pack} : (\Psi; \overrightarrow{Fr}) \multimap \exists \Gamma_1, \Gamma_2, \Gamma_3.(\Psi'; \overrightarrow{Fr}'; \textbf{void})} \tag{3.31}$$

$$\frac{\begin{array}{c}\ell \notin Dom(\Psi') \qquad \theta \notin Dom(\Gamma) \\ \Gamma \vdash I : (\Psi; \overrightarrow{Fr}) \multimap \exists \Gamma'.(\Psi'; \overrightarrow{Fr}'(\tau_0,\ldots,\tau_n); C\langle ? \rangle)\end{array}}{\begin{array}{l}\Gamma \vdash I\ \textbf{unpack}\ j : (\Psi; \overrightarrow{Fr}) \multimap \exists \Gamma', \ell{:}C, \theta. \\ \quad (\Psi', \ell \mapsto \theta; \overrightarrow{Fr}'(\tau_0,\ldots,\tau_{j-1}, \mathcal{Rep}_C\langle \theta \rangle, \tau_{j+1},\ldots,\tau_n); C\langle \ell \rangle)\end{array}} \tag{3.32}$$

$$\frac{\begin{array}{c}\Gamma \vdash I_1 : (\Psi; \overrightarrow{Fr}) \multimap \exists \Gamma_1.(\Psi_1; \overrightarrow{Fr}_1; \mathcal{Rep}_C\langle H \rangle) \\ \Gamma, \Gamma_1 \vdash I_2 : (ctx^+_{C,k}(H, \Psi_1); \overrightarrow{Fr}_1) \multimap \exists \Gamma'.(\Psi'; \overrightarrow{Fr}'; \tau) \\ \Gamma, \Gamma_1 \vdash I_3 : (ctx^-_{C,k}(H, \Psi_1); \overrightarrow{Fr}_1) \multimap \exists \Gamma'.(\Psi'; \overrightarrow{Fr}'; \tau)\end{array}}{\Gamma \vdash I_1\ I_2\ I_3\ \textbf{condst}\ k : (\Psi; \overrightarrow{Fr}) \multimap \exists \Gamma_1, \Gamma'.(\Psi'; \overrightarrow{Fr}'; \tau)} \tag{3.33}$$

$$\frac{\Gamma \vdash I_i : (\Psi_{i-1}; \overrightarrow{Fr}_{i-1}) \multimap \exists \Gamma_i.(\Psi_i; \overrightarrow{Fr}_i; \mathcal{Rep}_{C_i}\langle H_i \rangle)\ \forall i \in 1..n}{\begin{array}{l}\Gamma \vdash I_1\ \ldots\ I_n\ \textbf{newhist}\ C, k : (\Psi_0; \overrightarrow{Fr}_0) \multimap \exists \Gamma_1,\ldots,\Gamma_n. \\ \qquad (\Psi_n; \overrightarrow{Fr}_n; HC_{C,k}(\mathcal{Rep}_{C_1}\langle H_1 \rangle,\ldots,\mathcal{Rep}_{C_n}\langle H_n \rangle))\end{array}} \tag{3.34}$$

Figure 3.7 (Continued)

$$\frac{\begin{array}{c}\Gamma_1, \Gamma' \vdash I : (\Psi_1; \overrightarrow{Fr}_1) \multimap \exists \Gamma_2.(\Psi_2; \overrightarrow{Fr}_2; \tau) \\ \Psi'_1 \preceq \Psi_1 \quad \overrightarrow{Fr}'_1 \preceq \overrightarrow{Fr}_1 \quad \Psi_2 \preceq \Psi'_2 \quad \overrightarrow{Fr}_2 \preceq \overrightarrow{Fr}'_2 \quad \tau \preceq \tau'\end{array}}{\Gamma_1, \Gamma' \vdash I : (\Psi'_1; \overrightarrow{Fr}'_1) \multimap \exists \Gamma_2, \Gamma'.(\Psi'_2; \overrightarrow{Fr}'_2; \tau')} \tag{3.35}$$

$$\frac{}{\Gamma \vdash \boxed{\mathbf{0}} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; \mathbf{void})} \tag{3.36}$$

$$\frac{}{\Gamma \vdash \boxed{i4} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; \mathbf{int32})} \tag{3.37}$$

$$\frac{\Psi = \Psi' \star (\ell \mapsto H)}{\Gamma, \ell{:}C \vdash \boxed{\ell} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; C\langle \ell \rangle)} \tag{3.38}$$

$$\frac{}{\Gamma, \ell{:}C\langle ? \rangle \vdash \boxed{\ell} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; C\langle ? \rangle)} \tag{3.39}$$

$$\frac{}{\Gamma \vdash \boxed{\mathit{rep}_C(H)} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; \mathcal{R}\mathit{ep}_C\langle H \rangle)} \tag{3.40}$$

$$\frac{\Gamma \vdash I : (\Psi; \overrightarrow{Fr}) \multimap \exists \Gamma'.(\Psi'; \overrightarrow{Fr}' Fr_0; \tau)}{\Gamma \vdash I \ \mathbf{ret} : (\Psi; \overrightarrow{Fr}) \multimap \exists \Gamma'.(\Psi'; \overrightarrow{Fr}'; \tau)} \tag{3.41}$$

Mobile's typing rules are given in Figure 3.7. We provide some intuition behind each rule below. This intuition is elucidated in a more practical way in §3.3, which walks the type-checking algorithm through a sample Mobile program.

Rule 3.19 asserts that the **ldc.i4** instruction returns an integer and leaves the history map Ψ and frame list $\overrightarrow{Fr}$ unchanged.

Rule 3.20 assigns types to conditional instructions and will be discussed in depth because it is a good exemplar for understanding how control flow is modeled by the rest of the typing rules. A conditional's test expression I_1 is required to return an integer. Evaluating the test expression yields a new history map Ψ_1 and frame list $\overrightarrow{Fr}$, and this might yield additions Γ_1 to the typing context. All three of these flow into the typing judgments for the positive (I_2) and negative (I_3) branches of the conditional.

Terms I_2 and term I_3 are required to yield identical history maps Ψ' and frame lists $\overrightarrow{Fr}'$, as well as to make the same additions Γ' to the typing context. However, because Mobile types are conservative approximations of the security-relevant state of objects rather than exact representations, the above restriction does not require that two branches of a conditional exhibit the same sequences of security-relevant events. We require only that there exists some typing context, history map, and frame list that conservatively approximates the possible events that might have been exhibited by both branches. For example, if term I_2 assigns sequence e_2 to object ℓ and term I_3 assigns sequence e_3 to object ℓ, then one valid history map Ψ' for the result of the I_1 I_2 I_3 **cond** instruction assigns history abstraction $e_2 \cup e_3$ to ℓ, denoting that either e_2 or e_3 might have occurred.[2]

[2]A formal derivation of this would employ Rule 3.35. See the derivations in Appendix B for examples.

Rule 3.21 asserts that a **while** loop is well-typed if there exists a typing context, history map, and frame list that constitute a fixed point for the loop—that is, unrolling the loop once yields the same context, map, and list. This allows the type system to conservatively approximate any number of iterations of the loop. Rule 3.22 causes the typing contexts, history maps, and frame lists yielded by each term in a sequence to flow into the next term in the sequence. All terms in the sequence must yield results of type **void** except the last, whose type is taken as the type of the entire sequence.

Rules 3.23 and 3.24 assign types to instructions that load and store object fields. Fields can only be loaded from and stored to unpacked objects $C\langle\ell\rangle$, not packages. Also, unpacked objects cannot be stored directly to an object field since this would allow them to escape to the heap; they must be packed first. Rules 3.25 and 3.26 assign types to instructions that load and store argument slots in the current stack frame. Stack frames can store objects of any type, including unpacked objects.

Rule 3.27 assigns types to instructions that create new unpacked objects and initialize their fields. Each field initializer expression inherits the typing context, history map, and frame list yielded by the last because the operational semantics evaluates them in sequence. A new object is initially assigned the empty sequence ϵ as its trace. In order that this not constitute a security policy violation, we require that the empty sequence be a prefix of the policy assigned to that object's class, i.e. $\epsilon \in \mathit{pre}(\mathit{policy}(C))$. Observe that this is always true except when $\mathit{policy}(C) = \emptyset$.

Rule 3.28 assigns types to method calls. After evaluating all the arguments, a typing context $\Gamma_0, \ldots, \Gamma_n$, a history map Ψ_{n+1}, and a frame list $(\tau_0, \ldots, \tau_n)$ result. The typing rule requires that the called method's formal signature *Sig*

must alpha-vary[3] (denoted by the <: operator) to some signature of the form $(\Psi_{in}, (\tau_0, \ldots, \tau_n)) \multimap \exists\Gamma_{out}.(\Psi_{out}, Fr_{out}, \tau)$ where Ψ_{in} is the part of the caller's history mapping Ψ_{n+1} that flows into the callee and remains in-scope within the method. This alpha-variance corresponds to asserting that there must be a way to consistently rename callee typing variables to caller typing variables. The history map that results includes objects that were in-scope in the callee (Ψ_{out}) and that may have been modified, as well as those that remained out-of-scope (Ψ_{unused}).

Rule 3.29 assigns types to instructions that exhibit events. The exhibited event is appended to the history abstraction that approximates the object's state, and we require that this new history abstraction is a prefix of the class's security policy. Thus, a Mobile program that exhibits a trace that violates the security policy is not well-typed.

Rules 3.30 and 3.31 assign types to instructions that create and pack packages. Rule 3.30 introduces a fresh type variable ℓ to refer to the new package. Rule 3.31 packs an object of type $C\langle\ell\rangle$ into a package of type $C\langle?\rangle$. We require that the runtime state value $\mathcal{R}ep_C\langle H'\rangle$ passed to the **pack** instruction must be an accurate approximation of the object's state H; i.e., $H \subseteq H'$. Additionally, the object's state must satisfy the policy since packed objects can escape to the heap and therefore might persist, untracked, until the program's termination. Observe that the object identity variable ℓ for the packed object disappears from the history map Ψ' so that any future direct references to the object are not well-typed.

Rule 3.32 unpacks a package, yielding an object of type $C\langle\ell\rangle$ and a runtime state value of type $\mathcal{R}ep_C\langle\theta\rangle$. Since the object's state is unknown, it is assigned

[3]Formally, we say that a signature Sig_1 *alpha-varies* to signature Sig_2 in context Γ, written $\Gamma \vdash Sig_1 <: Sig_2$, if there exists a substitution $\sigma : \ell \to \ell$ such that $\sigma(Sig_1) = Sig_2$ and any free variables in Sig_2 are drawn from Γ.

a fresh history variable θ. The new object history map assigns θ to ℓ. Typing Rules 3.33 and 3.34 for testing and creating runtime state values are discussed in §3.2.4.

Rule 3.35 allows weakening of types so that suitable types can be inferred at code join points. (See the discussion of Rule 3.20 above.) In particular, a term's type can be weakened by reducing items to the left of the linear implication ($\multimap$) and enlarging items to the right of the linear implication, according to subtyping relation $\preceq$ defined in Figure 3.3.

Rules 3.36 – 3.41 assign values to terms introduced by the small-step operational semantics. Rules 3.36 – 3.40 assign the obvious types to terms that have been reduced to values, and Rule 3.41 requires that **ret** instructions always be performed in settings where the stack is non-empty.

3.2.4 History Module Plug-ins

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 $\mathcal{R}\!\mathit{ep}_C\langle H\rangle$ of a history value constructed using constructor k for security-relevant class C, and we define $ctx^{+}_{C,k}(H, \Psi)$ and $ctx^{-}_{C,k}(H, \Psi)$ 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

$$test_{C,k}(H) = 0 \Longrightarrow \Psi \preceq ctx^{-}_{C,k}(H,\Psi)(\ell) \tag{3.42}$$

and

$$test_{C,k}(H) \neq 0 \Longrightarrow \Psi \preceq ctx^{+}_{C,k}(H,\Psi)(\ell) \tag{3.43}$$

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 $\mathcal{Rep}_{C_1}\langle H_1\rangle, \ldots, \mathcal{Rep}_{C_n}\langle H_n\rangle$ such that $n = arity(HC_{C,k})$, there exists some H such that

$$HC_{C,k}(\mathcal{Rep}_{C_1}\langle H_1\rangle, \ldots, \mathcal{Rep}_{C_n}\langle H_n\rangle) = \mathcal{Rep}_C\langle H\rangle \tag{3.44}$$

and

$$hc_{C,k}(\mathit{rep}_{C_1}(H_1), \ldots, \mathit{rep}_{C_n}(H_n)) = \mathit{rep}_C(H) \tag{3.45}$$

Typing Rules 3.33 and 3.34 of Figure 3.7 make use of the above definitions. Rule 3.33 refines the history map according to functions $ctx^{+}_{C,k}$ and $ctx^{-}_{C,k}$ in branches of a conditional that tests runtime state values. Rule 3.34 assigns the type defined by function $HC_{C,k}$ to new runtime state values.

A *history module plug-in* provides definitions of $hc_{C,k}$, $test_{C,k}$, $HC_{C,k}$, $ctx^{+}_{C,k}$, and $ctx^{-}_{C,k}$ that satisfy the above requirements. Such a plug-in can extend Mobile to support different schemes for representing runtime state and thereby enforce different security policies. Multiple plug-ins can also be used simultaneously to enforce different classes of security policies for different security-relevant classes in a single program. Many different plug-ins are possible, but three particularly useful ones are described below.

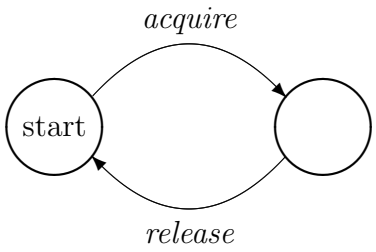


Figure 3.8: A DFA for an access protocol policy

Access Protocol Policies Deterministic finite automata (DFA's) [HU79] model security policies by accepting those sequences of symbols that correspond to permissible event sequences. Such automata can describe security policies that prescribe protocols for accessing system resources. For example, a protocol that requires a certain resource to be acquired before it is released could be modeled by the two-state DFA depicted in Figure 3.8.

A history module plug-in that supports security policies expressed as DFA's could implement runtime state values as integers, and define

$$hc_{C,k}() =_{\text{def}} k$$

$$test_{C,k}(\mathit{rep}_C(\theta)) =_{\text{def}} \begin{cases} 1 & \text{if } \mathit{rep}_C(\theta) = k \\ 0 & \text{otherwise} \end{cases}$$

$$ctx^{+}_{C,k}(\theta, \Psi) =_{\text{def}} \Psi[\theta \mapsto \theta \cap H_k]$$

$$ctx^{-}_{C,k}(\theta, \Psi) =_{\text{def}} \Psi[\theta \mapsto \theta \cap (\cup_{i \neq k} H_i)]$$

for each integer k and all classes C, where H_k is a closed history abstraction statically assigned by the policy-writer to each integer constant k. This scheme allows a Mobile program to represent object security states at runtime using a DFA in the following way: The policy-writer first assigns to each state of the DFA an integer constant k. He next defines each closed history abstraction H_k

to be the set of traces that can cause the DFA to arrive in state k. The Mobile program would then implement tests **condst** of runtime state values as equality comparisons between the integer runtime state value to be tested and an integer constant to determine at runtime if the DFA was in any given state.

The assignments of closed history abstractions H_k to integers k are not trusted, so such a mapping can be defined by the Mobile program itself (e.g., in settings where self-monitoring programs are produced by a common rewriter or where separately produced programs do not exchange objects) or by the policy-writer (in settings where the mapping must be defined at a system global level for consistency between programs).

Resource Bound Policies Another class of useful security policies are those that bound the number of times a resource can be used. When the number of allowed uses is large, counters are a more efficient model of such policies than DFA's. (For example, when the number of allowable uses is 2^{30}, a 4-byte integer counter can model the policy, whereas the smallest suitable DFA has 2^{30} states.)

A history module plug-in that supports resource bound policies could implement runtime state values as integers, and define

$$hc_{C,k}() =_{\text{def}} k$$

$$test_{C,k}(\mathit{rep}_C(\theta)) =_{\text{def}} \begin{cases} 1 & \text{if } \mathit{rep}_C(\theta) \leq k \\ 0 & \text{otherwise} \end{cases}$$

$$ctx^{+}_{C,k}(\theta, \Psi) =_{\text{def}} \Psi[\theta \mapsto \theta \cap (\cup_{i \leq k} e^i)]$$

$$ctx^{-}_{C,k}(\theta, \Psi) =_{\text{def}} \Psi[\theta \mapsto \theta \cap e^{\omega}]$$

for each integer k and all classes C, where e is the event that corresponds to using the resource.

A runtime state value would therefore constitute an integer upper bound on the number of times an associated object has been used so far during the program's execution. The Mobile program would then implement tests **condst** of runtime state values as inequality comparisons between the integer runtime state value to be tested and an integer constant to determine at runtime if the resource was potentially near its usage bound.

Exact Traces A history module plug-in can even track every object's exact trace at runtime, though this would be computationally expensive in practice. Although this is impractical in some settings, we show a formalism for doing so below in order to demonstrate the power of Mobile's type system.

The set of possible Mobile events is finite because events are program operations, and real computer architectures only have a finite number of possible instructions. Letting n be the number of possible events and letting $e_1, \ldots, e_n$ be the names of the events, we define[4]

$$hc_{C,0}() =_{\text{def}} obj_{null}\{\}$$

$$hc_{C,k}(\ell) =_{\text{def}} obj_{list}\{evt = k, next = \ell\} \quad \forall k \in 1..n$$

$$test_{C,\epsilon}(\mathit{rep}_C(\theta)) =_{\text{def}} \begin{cases} 1 & \text{if } \mathit{rep}_C(\theta) = obj_{null}\{\} \\ 0 & \text{otherwise} \end{cases}$$

[4]Here we index history plug-in *test* functions by finite-length event sequences instead of by integers for notational simplicity. This is isomorphic to an integer indexing because the set of finite-length event sequences is countable.

$$test_{C,\overrightarrow{e}\,e_k}(\mathit{rep}_C(\theta)) =_{\text{def}} \begin{cases} 1 & \text{if } \mathit{rep}_C(\theta) = obj_{list}\{evt = k, next = \ell\} \\ & \text{and } test_{C,\overrightarrow{e}}(\ell) = 1 \\ 0 & \text{otherwise} \end{cases}$$

$$ctx^{+}_{C,\overrightarrow{e}}(\theta, \Psi) =_{\text{def}} \Psi[\theta \mapsto \theta \cap \overrightarrow{e}]$$

$$ctx^{-}_{C,\overrightarrow{e}}(\theta, \Psi) =_{\text{def}} \Psi$$

That is, history constructor $hc_{C,0}()$ returns a runtime representation of the empty sequence (as an empty linked list), and history constructor $hc_{C,k}(\ell)$ for $k \in 1..n$ takes an existing sequence representation and appends event e_k to it (by prepending integer k to linked list ℓ). Using these two constructors, a Mobile program can construct a runtime state value that represents any finite-length event sequence. The program can use the $test_{C,\overrightarrow{e}}$ operation to compare an object's runtime state value to any particular event sequence $\overrightarrow{e}$. In this way, a Mobile program could precisely record and monitor every object's event trace at runtime.

The above shows that history module plug-ins are an extremely powerful and versatile way to extend the Mobile type system to support useful classes of security policies. By keeping the history module plug-in implementation abstract in the formal treatment in this chapter, we allow Mobile to be specialized to suit various applications and architectures.

3.3 An Example Mobile Program

To indicate how the operational semantics and typing rules described in the previous sections play out in practice, we here informally walk the type-checking algorithm through the sample Mobile program given in Figure 3.9.

```
1    (newobj C()) starg 1;
2    (ldarg 1) evt e1;
3    (ldarg 1) evt e2;
4    (newpackage C) starg 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 e1) (...) condst C, 0
```

Figure 3.9: Sample Mobile program

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 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, 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 e_1 and e_2 on the object in local register 1, the type-checker's new history map would satisfy $\Psi'(\ell) = e_1 e_2$. At each point that a security-relevant event is performed, the type system ensures that the new trace satisfies a prefix of the security policy. For example, when type-checking line 3, the type-checker would verify that $e_1 e_2 \subseteq \mathit{pre}(\mathit{policy}(C))$, 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 is 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 $\mathcal{Rep}_C\langle H\rangle$, 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. Since only packed objects can leak to the heap, the restriction that packed objects must be in a policy-adherent state is a potential limitation of the type system. That is, it might often be desirable to leak an object that is not yet in a policy-adherent state to the heap, but later retrieve it and restore it to a

policy-adherent state before the program terminates. In Chapter 4 we show how Mobile implementations can use finalizer code to avoid this restriction and leak objects to the heap even when they are not yet in a policy-adherent state.

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 the code 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 $\mathcal{R}ep_C\langle\theta\rangle$, respectively, and the new history map satisfies

$\Psi'(\ell') = \theta$. The type $C\langle ? \rangle$ of a package can hence be thought of as an existential type binding type variables ℓ' and θ.

If the sample program were at this point to perform security-relevant event e on the newly unpacked object, Mobile's type system would reject because it would be unable to statically verify that $\theta e \subseteq \mathit{policy}(C)$ (since nothing is statically known about history θ)[5]. 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 $\mathcal{Rep}_C\langle e_1 e_2\rangle$, then in the positive branch of the conditional, the type system can statically infer that $\theta = e_1 e_2$. If $\mathit{policy}(C) = (e_1 e_2)^\omega$, then a Mobile program could execute I **evt** e_1 within the positive branch of such a conditional (where I is the object that was unpacked), because $e_1 e_2 e_1 \subseteq \mathit{pre}((e_1 e_2)^\omega)$; but the type-checker would reject a program that executed I **evt** e_2 in the positive branch, since $e_1 e_2 e_2 \not\subseteq \mathit{pre}((e_1 e_2)^\omega)$.

Suppose that history value constructor **newhist** $C, 0$ takes no arguments and yields a runtime value that represents history $e_1 e_2$; and suppose that conditional test **condst** $C, 0$ compares a runtime state value to the value that represents history $e_1 e_2$. Formally, suppose that $HC_{C,0}() = \mathcal{Rep}_C\langle e_1 e_2\rangle$ and $ctx^+_{C,0}(\theta, \Psi) = \Psi[\theta \mapsto e_1 e_2]$. Thus, in the positive branch of such a test, the type-checker's object history map can be refined by substituting $e_1 e_2$ for any instances of the history variable

[5]In actuality, one could safely infer that $\theta \subseteq \mathit{policy}(C)$ holds because θ came from a package, and all packed objects satisfy the security policy. It would then suffice for the type system to verify that $\mathit{policy}(C)\, e \subseteq \mathit{policy}(C)$ holds. However, to keep our subset and subtyping relations simple, we have chosen not to reflect this refinement in the typing rules presented in this chapter.

being tested. Then if $policy(C) = (e_1 e_2)^\omega$, 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 $e_1 e_2$, and therefore it is safe to perform event e_1 on it. However, if $policy(C) = e_1 e_2 e_2$, then the type-checker would reject, because $e_1 e_2 e_1$ is not a prefix of $e_1 e_2 e_2$.

In the negative branch of this conditional the type-checker can infer that the object in local register 4 has a history represented by a state value other than the one that it was tested against. The history map could therefore be refined by substituting history variable θ with the union of all of the history abstractions associated with all of the other possible runtime state values defined for that object's class.

3.4 Policy Adherence of Mobile Programs

The operational semantics of Mobile presented in §3.2.2 permit untyped Mobile programs to enter bad terminal states—states in which the Mobile program has not been reduced to a value but no progress can be made. 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). Mobile's type system presented in §3.2.3 prevents both policy violations and bad terminal states, except that it does not prevent **unpack** 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. We prove below that Mobile programs well-typed with respect to a security policy will not violate the security policy when executed even if they enter a bad state.

Formally, we define well-typed by

Definition 3.1. A method $C{::}m.Sig$ with signature $Sig = \forall\Gamma_{in}.(\Psi_{in}, Fr_{in}) \multimap \exists\Gamma_{out}.(\Psi_{out}, Fr_{out}, \tau)$ 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 term I is defined by $I = methodbody(C{::}m.Sig)$.

Definition 3.2. 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 $C_{main}{::}main.Sig_{main} \in Dom(methodbody)$ having a method signature of the form $Sig_{main} = \forall\Gamma_{in}.(\Psi_{in}, (\tau_1, \ldots, \tau_n)) \multimap \exists\Gamma_{out}.(\Psi_{out}, Fr_{out}, \tau_{out})$ such that for all substitutions $\sigma : \theta \rightarrow \overrightarrow{e}$ 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 3.2 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 normally, 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 3.3 (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 3.4 (Prefix Adherent)**.** A heap h is *prefix-adherent* if, for all class objects $obj_C\{\ldots\}^{\overrightarrow{e}} \in Rng(h)$, $\overrightarrow{e} \subseteq pre(policy(C))$.

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 all object fields and stack slots in ψ have values of appropriate types, and the heap in ψ is prefix-adherent. (See Appendix B for a formal definition.) The following two theorems then establish that well-typed Mobile programs do not violate the security policy.

Theorem 3.1 (Terminating Policy Adherence)**.** *Assume that a Mobile program is well-typed, and that, as per Definition 3.2, 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* $\psi, methodbody(C_{main}::main.Sig) \rightsquigarrow^* (h', s'), \boxed{v}$ *holds, then* h' *is policy-adherent.*

Proof. See Appendix C. □

Theorem 3.2 (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* $(h, s), I \rightsquigarrow^n (h', s'), I'$ *holds, then* h' *is prefix-adherent.*

Proof. See Appendix C. □

An important consequence of both 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 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.

3.5 Related Work

Type-systems $\lambda_{\mathcal{A}}$ [Wal00] and λ_{hist} [SS04] 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 such as 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 enforces policies that are universally quantified over objects of any given class, and therefore allows objects to be treated as first-class in policy specifications.

PCC has been proposed as a framework for supporting certifying rewriting using temporal logic [BL02]. The approach is potentially powerful, but does not presently support languages that include exceptions, concurrency, and other features found in real programming languages [Ber04, p. 173]. It is therefore unclear whether proof size and verification speed would scale well in practical settings.

CQual [FTA02] and Vault [DF01] are C-like languages that enforce history-based properties of objects by employing a flow-sensitive type system based on alias types [SWM00] and typestates [DF04b]. Security-relevant objects in CQual or Vault programs have their base types augmented with type qualifiers, which statically track the security-relevant state of the object. A type-checker then determines 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. The Vault type-checker identifies instructions that test these state values to ensure that those tests will prevent security violations when the program is executed.

Fugue [DF04a] is a static verifier based on Vault that uses programmer-supplied specifications to find bugs in .NET source code. It verifies policies that constrain the use of system resources or that prescribe protocols that constrain the order in which methods may be called on objects. Fugue supports any source language that compiles to managed .NET CIL code, but it does not support exceptions, finalizers, or concurrency. It additionally lacks a formal proof of soundness for its aliasing analysis and type system.

CQual, Vault, and Fugue 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 are shared between threads.

In this chapter, we have therefore taken the approach of L^3 [MAF05], wherein linearly-typed items are permitted to leak to the heap by packing them into packages—shared data structures with limited interfaces. As with ownership types [CNP01, CD02], 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.

3.6 Conclusions and Future Work

Mobile's type system and the theorems presented in §3.4 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. The language of security policies is left abstract and could consist of DFA's, LTL expressions, or any computable language of finite and infinite event sequences.

Our work 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 dynamic 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. Type systems for coordinated data structures [RG05] could potentially be leveraged to enforce invariants over these decoupled objects and states.

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 because there is a natural rewriting strategy (outlined in §3.1.3) whereby well-typed Mobile code can be automatically generated from untrusted CIL code. A more powerful type system could employ a richer language like Hoare Logic [Hoa69] to track data-sensitive control flow. This could allow clever rewriters to eliminate additional runtime checks by statically proving that they are unnecessary. However, formulating a sound and complete Hoare Logic for .NET that includes objects and concurrency is challenging; furthermore, the burden of producing useful proofs in this logic would be pushed to the rewriter. Future work should investigate rewriting strategies that could make such an approach worthwhile.

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.

Chapter 4

Implementation of Mobile

This chapter includes previously published [HMS05, HMS06a] joint work with Greg Morrisett and Fred B. Schneider.

4.1 Overview

In this chapter we describe a prototype implementation of Mobile for the Microsoft .NET Framework. Our implementation consists of a program-rewriter, which transforms .NET CIL bytecode programs into Mobile programs according to a declared security policy, and a type-checker, which verifies that Mobile programs are well-typed with respect to a security policy. Mobile programs are implemented as annotated CIL bytecode programs, where the annotations are encoded as CIL custom attributes and as CIL bytecode instructions. Mobile programs are therefore legal CIL programs that can be executed without modification by a .NET virtual machine once they are verified by the type-checker. Annotations can also be automatically stripped from from Mobile programs after type-checking to produce smaller binaries.

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

The implementation expresses security policies as star-free ω-regular expressions[1] [Lad77, Tho79] over this language of events, where each such expression is universally quantified by object class. That is, a security policy specifies a set of security-relevant classes, and a star-free ω-regular expression for each, defining the set of permitted method call sequences that may be invoked on any instance of that class at runtime. Formally, our implementation enforces policies induced by detectors of the form

$$\hat{\mathcal{P}}(\chi) = (\forall o : o \in C_1 : \chi_o \in W_1) \wedge \cdots \wedge (\forall o : o \in C_n : \chi_o \in W_n)$$

where each $C_1, \ldots, C_n$ is an object class, each $W_1, \ldots, W_n$ is a star-free ω-regular expression, and the notation χ_o denotes the subsequence of execution χ consisting of method calls exhibited on object o.

Our implementation does not support policy specifications in which two or more of $C_1, \ldots, C_n$ are related by inheritance. If a security policy declares a given class to be security-relevant, then it implicitly declares all of that class's subclasses to also be security-relevant and restricted to the same set of method call sequences as their parent class. Policies regard method names rather than the methods themselves, so that a policy that constrains calls to method $C_1{::}m$ also constrains calls to method $C_2{::}m$ when class C_2 is a subclass of class C_1. A facility to specify different event sequence sets for different security-relevant classes related by inheritance is left to future work.

The type-checker implementation supports the full managed subset of the .NET CIL (minus reflection), including exceptions, finalizers, and multithreading. Support for finalization is particularly useful because it affords Mobile programs a

[1]Star-free ω-regular expressions are regular expressions except with ω to denote finite or infinite repetition instead of Kleene star to denote strictly finite repetition.

means to overcome the limitation described in §3.3 that required all packed objects to be in a policy-adherent state. In an implementation that supports finalizers, packed objects need not be in a policy-adherent state. Instead, they must be in a state that satisfies their finalizer's precondition, and the finalizer's postcondition must satisfy the security policy. These requirements ensure that a Mobile program's finalizer code will bring any packed object into a policy-adherent state before termination. This allows Mobile programs to enforce security policies lazily by including clean-up code in finalizers instead of bringing every object into a policy-adherent state whenever it might escape to the heap.

Runtime state values are implemented as integers. Thus, our **newhist** operation is simply an **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. Tests of runtime state values consist of equality comparisons with integer constants in the manner described in §3.2.4 for implementing access protocol policies. This implementation suffices to support IRM's that model security policies as deterministic, finite-state security automata.

Packages are implemented as a small trusted class written in managed C#. Our **pack** and **unpack** instructions are calls to methods in that class. A security policy can identify zero or more package implementations that a rewriter can use to safely leak security-relevant objects to the heap.

The chapter proceeds as follows. In §4.2 we describe the policy specification language. Then §4.3 and §4.4 describe the type-checker and rewriter implementations, respectively. Finally, §4.6 discusses related work and §4.7 proposes future work.

4.2 Security Policy Specifications

Security policy specifications consist of a list of two kinds of declarations: *class declarations* and *event declarations*.

Class declarations Class declarations identify security-relevant classes that the type system should track, and describe the sequences of events that Mobile programs may exhibit on each such object. They have the syntax

$$\langle classname \rangle \{ \langle numevents \rangle, \langle packinvariant \rangle, \langle classpolicy \rangle, \langle runtimestatevalues \rangle \}$$

where $\langle classname \rangle$ is a fully qualified class name identifying a class in the .NET namespace hierarchy, $\langle numevents \rangle$ is an integer denoting the number of different security-relevant operations for objects of this class, $\langle packinvariant \rangle$ is a star-free ω-regular expression denoting the security state an object must have whenever it is packed, $\langle classpolicy \rangle$ is a star-free ω-regular expression denoting the set of permissible event sequences for objects of this class, and $\langle runtimestatevalues \rangle$ specifies the list of runtime state values (i.e., integers) Mobile programs may use and what state abstraction each represents.

The $\langle runtimestatevalues \rangle$ specification is not trusted but is included in the policy statement to support separate rewriting of .NET assemblies. That is, if the same security policy is to be enforced over multiple .NET assemblies, and the assemblies might share packages, then the assemblies are required to use the same runtime representations of state abstractions. Policy-writers can omit the $\langle runtimestatevalues \rangle$ field from a policy specification to cause the program-rewriter to automatically generate one that is suitable for the $\langle classpolicy \rangle$. (This is achieved by converting the $\langle classpolicy \rangle$ into a DFA that accepts the set of finite prefixes of $\langle classpolicy \rangle$, and then using the procedure described in §3.2.4 to

convert this DFA into a history plug-in.) Thus, when rewriting a set of assemblies according to the same policy, the first invocation of the rewriter causes a suitable ⟨*runtimestatevalues*⟩ specification to be selected and the remaining invocations reuse the same specification.

The following is an example class declaration:

$$[\texttt{mscorlib}]\texttt{System.IO.File}\{3, , (\texttt{e1}\hat{}\texttt{e2}*\hat{}\texttt{e3})*, [\,]\}$$

This declaration identifies the `File` class provided by the .NET runtime system as a security-relevant class with 3 events. Since the class has no finalizer, the ⟨*packinvariant*⟩ is omitted, requiring packed instances of this class to satisfy the ⟨*classpolicy*⟩. In the ⟨*classpolicy*⟩, $*$ represents ω, $\hat{}$ represents concatenation, $+$ represents disjunction, # and `0` represent the empty sequence and the empty set, and events are numbered `e1`, `e2`, etc. The ⟨*runtimestatevalues*⟩ spec is omitted, causing the rewriter to choose a suitable collection of runtime state values. In this case, the rewriter could choose the 3-state DFA encoded by the following ⟨*runtimestatevalues*⟩ specification:

$$\begin{aligned}[\ &\texttt{s0} = (\texttt{e1}\hat{}\texttt{e2}*\hat{}\texttt{e3})*,\\ &\texttt{s1} = (\texttt{e1}\hat{}\texttt{e2}*\hat{}\texttt{e3})*\hat{}\texttt{e1},\\ &\texttt{s2} = (\texttt{e1}\hat{}\texttt{e2}*\hat{}\texttt{e3})*\hat{}\texttt{e1}\hat{}\texttt{e2}\hat{}\texttt{e2}*\ ]\end{aligned}$$

That is, it could use integer 0 to represent objects in a policy-adherent state, 1 to represent objects that have exhibited event $e1$, and 2 to represent objects that have exhibited event $e1$ followed by one or more of event $e2$. The DFA that this specification encodes is shown in Figure 4.1.

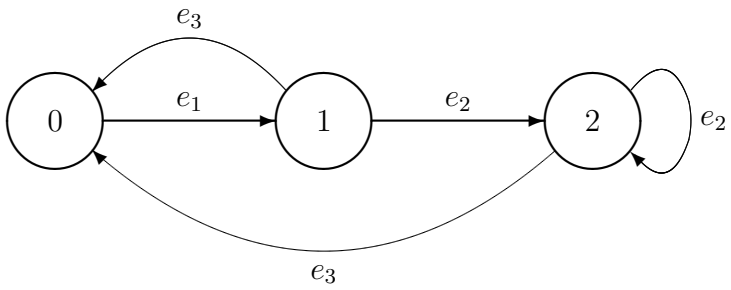


Figure 4.1: A DFA accepting $(e_1 e_2^* e_3)^*$

Event Declarations Event declarations identify method calls as security-relevant events. They have the syntax

$$\langle classname \rangle.\langle methodname \rangle\ \langle precondition \rangle(\langle arguments \rangle)\langle postcondition \rangle$$

where $\langle precondition \rangle$ encodes typing context Γ_{in} and history map Ψ_{in} defined in Figure 3.2 (i.e., it binds any type variables appearing in the $\langle arguments \rangle$), and $\langle postcondition \rangle$ encodes a typing context Γ_{out} and history map Ψ_{out} for a normal return from the method and for various exceptions that the method might throw. The $\langle arguments \rangle$ provide the types of the method's arguments, including any implicit arguments like the object's `this` pointer.

For example, the event declaration

```
[mscorlib]System.IO.File.Open
    [o1 = [mscorlib]System.IO.File : h1]
    (C<o1>, _, _)
    [ Ret([o1 = [mscorlib]System.IO.File : h1^e1]_),
      Exn([o1 = [mscorlib]System.IO.File : h1]_) ]
```

identifies the `Open` method as a security-relevant method call for objects of class `System.IO.File`. This method accepts an unpacked, security-relevant class `C<o1>`

as its first argument (along with two other security-irrelevant arguments denoted by underscores), where `o1` is a type variable that refers to an object of type `System.IO.File` in some state `h1`. (`h1` is a history variable that is implicitly universally quantified over the universe of all event sequences.) If the method returns normally, object `o1` will be in state `h1^e1`, denoting that security-relevant event `e1` has been appended to its trace. If the method throws an exception, object `o1` remains in state `h1`. The underscore at the end of the `Ret(...)` part of the specification denotes the type of the return value, which is in this case security-irrelevant. The underscore at the end of the `Exn(...)` part refers to the type of the exception thrown, which is also security-irrelevant.

Aside from untracked objects (denoted by underscores) and unpacked, security-relevant classes (denoted by `C<o1>`), event declarations can also refer to runtime state values (denoted by `H<classname : h1>`) and managed pointers to any of the above (denoted by prepending an ampersand to the type).

Event declarations can also be used to identify the method calls that implement Mobile's **pack** and **unpack** operations. This will be discussed further in §4.4.

4.3 Type-checking Algorithm

The type-checker was written in Ocaml and uses Microsoft's .NET ILX SDK [Sym01] to read and manipulate .NET bytecode binaries. We also use Conchon, Filliâtre, and Signoles's `ocamlgraph` library [CFS06] to construct and manipulate DFA's. Aside from these libraries and the OCaml runtime libraries, the type-checker consists of about two thousand lines of Ocaml code. About half of that code duplicates portions of the .NET bytecode verification algorithm, so we spec-

ulate that adding the type-checker to an existing .NET system would contribute about one thousand lines of code to the trusted computing base.

The type-checker accepts as input a .NET bytecode program and a policy specification of the form described in §4.2 and returns with success if the annotations in the bytecode program suffice to prove that the program satisfies the security policy. Otherwise it signals a rejection by printing an error message. The type-checking algorithm consists of walking over each instruction in a linear fashion and applying the Mobile typing rules given in §3.2.3 to decide whether the code is well-typed with respect to the security policy. In addition to the control flow constructs modeled by Mobile, the type-checker also tracks control flow in the presence of exceptions and the various CIL branching instructions.

4.3.1 Annotations

To keep the type-checking algorithm tractable, the type-checker recognizes three kinds of (untrusted) typing annotations that aid in the type-inference.

- At each join point, the type-checker expects a *virtual machine state annotation* that abstracts the security-relevant state that will exist whenever execution reaches that program point.

- Each method that accepts or returns unpacked classes must be annotated with its function signature *Sig* (see Figure 3.2).

- Each call site involving an unpacked class must be annotated with an instantiation of callee type variable names as caller type variable names.

An example of each kind of annotation is given in Figure 4.2. Virtual machine state annotations describe the types of values in the current argument frame, local

A virtual machine state annotation:

$$\{\ \texttt{args} = [\texttt{C} < \texttt{o1} >, \_, \_, \texttt{H} < \texttt{h1} >],$$

$$\texttt{locs} = [\_, \_, \texttt{C} < \texttt{o2} >]$$

$$\texttt{stk} = [\_, \&\texttt{H} < \texttt{o2} >]$$

$$\texttt{ctx} = [\ \texttt{o1} = [\texttt{mscorlib}]\texttt{System.IO.File} : \texttt{h1},$$

$$\texttt{o2} = [\texttt{mscorlib}]\texttt{System.IO.File} : \texttt{h2},$$

$$\texttt{h1} = \#,$$

$$\texttt{h2} = \texttt{e1}\hat{}\texttt{e2}\ ]$$

$$\texttt{exnstate} = \texttt{None}\ \}$$

A method annotation:

$$[\ \texttt{o1} = [\texttt{mscorlib}]\texttt{System.IO.File} : \texttt{h1}\ ]$$

$$(\texttt{C}<\texttt{o1}>, \_, \_, \texttt{H}<[\texttt{mscorlib}]\texttt{System.IO.File} : \texttt{h1}>)$$

$$[\ \texttt{Ret}([\texttt{o1} = [\texttt{mscorlib}]\texttt{System.IO.File} : \texttt{h1}\hat{}\texttt{e1}]\_)\ ]$$

A call site annotation:

$$[\ \texttt{h1} = \texttt{e1}\hat{}\texttt{e2}, \texttt{h2} = \texttt{h62}\ ]$$

Figure 4.2: Mobile annotations

variable frame, and evaluation stack, as well as describing the current typing context and history map. The `exnstate` field gives the type of the current exception being thrown[2] if the annotation lies within an exception handler, or `None` otherwise. Method annotations have the same syntax as described for class declarations in §4.2 except without the class name and method name. Call site annotations instantiate each callee history variable either by assigning it a closed history expression or by renaming it to a caller history variable. (Object type variable renaming is not included in the annotation because it is inferred automatically by the type-checker.)

The type-checker verifies the correctness of each annotation. Virtual machine state annotations are correct if the state inferred by the type-checker along each control flow path that arrives at the annotated join point is a subset of the state described by the annotation. Method annotations are correct if the type inferred for the body of the method is a subset of the type given by the annotation. Finally, call site annotations must match both the signature of the callee method and the states inferred for all control flow paths reaching the call site.

Method annotations are implemented by attaching a CIL custom attribute to each annotated method. The other kinds of annotations refer to code points within methods, so they cannot be implemented in this way. One way to implement them would be to attach custom attributes to the methods with a numerical index identifying which instruction in the method's body they annotate. However, the ILX SDK library does not currently expose bytecode indexes, so for the prototype, we implemented the remaining annotations by adding the two-instruction sequence

[2]An exact type for exceptions cannot always be determined statically. In those cases, a supertype such as `System.Exception` is inferred.

```
ldstr ⟨annotation⟩

pop
```

to each annotation point, which loads a string constant onto the evaluation stack and pops it back off again. These extra instructions won't affect execution, but can be automatically stripped from the binary along with method annotations after type-checking is complete to produce smaller and faster binaries.[3]

4.3.2 Subset Relations

To correctly apply typing Rule 3.27, 3.29, 3.31, and 3.35 given in Figure 3.7, the type-checker must verify subset relations over the language of history abstractions given in Figure 3.2. That is, it must decide subset relations over the language of star-free ω-regular expressions plus variables and intersection.

Although deciding subset for this language is not tractable in general, the task is simplified by observing that real Mobile code only introduces history variables at the beginnings of expressions (e.g., when an unpacked object flows into a method whose signature assigns a name to the object's state) and only introduces intersections that involve a variable and a closed history abstraction (e.g., when a runtime state value is tested). We show in Appendix D that the resulting sub-language is decidable. In particular, it can be reduced to the subset problem for regular expressions.

Our type-checker implementation uses homogeneous non-deterministic automata (hNFA's) [Cha01] to model regular expressions. It decides relation $L(N_1) \subseteq L(N_2)$ (where the notation $L(N)$ denotes the language accepted by hNFA N) by

[3]In practice we have not observed any experiments in which these annotations noticeably affected the size or runtime speed of the binary, probably because the JIT compiler optimizes them away.

deciding relation $L(N_1) \cap \overline{L(N_2)} \neq \emptyset$. That is, the second hNFA is complemented, intersected with the first hNFA, and the final states of the resultant NFA are tested for reachability. Complementation is achieved by using the `ccp` algorithm [CCP04] to determinize the hNFA, and then using the algorithms described in [HU79] for complementing a DFA and intersecting two automata.

Although this algorithm is exponential in the size of N_2 in the worst case, it is linear in the size of N_1. Untrusted code can dictate N_1 but not N_2 because because the right-hand sides of the subset relations in the typing rules in Figure 3.7 are determined by the security policy, not the untrusted code. Typing Rule 3.29 allows untrusted code to cause the type-checker to build up an hNFA N_1 that is at most linear in the size of the code. Thus, our implementation of the type-checking algorithm is quadratic in the size of the code being type-checked, but exponential in the size of the hNFA that models the security policy. Despite this exponential complexity, in practice we have found the type-checking algorithm to exhibit reasonable runtimes (see §4.5).

4.4 Program-rewriting Algorithm

Our rewriter implementation was also written in Ocaml and uses the same runtime libraries as the type-checker. Discounting these runtime libraries, it consists of about 3500 lines of Ocaml code. The rewriter accepts as input an arbitrary managed CIL bytecode program and a policy specification of the form described in §4.2 and outputs a new CIL bytecode program that has been rewritten and annotated according to the security policy. The rewriter supports a significant portion of the managed subset of the .NET CIL (minus reflection), but does not support code that accesses security-relevant methods through delegates (managed

function pointers), does not support policies that declare package classes or their superclasses to be security-relevant (e.g., the `System.Object` class may not be identified as a security-relevant class), and does not support code that throws security-relevant objects as exceptions. Also, the behavior of some multithreaded programs is not preserved; this limitation will be discussed in more detail later.

Our rewriter adopts the rewriting strategy outlined in §3.1.3; it changes all instances of security-relevant classes to packages. Whenever a field is accessed, the rewriter inserts instructions to first unpack the package, then access the unpacked class's field, and finally repack the object. Method calls that are security-relevant events are redirected to wrapper methods that unpack the package, test its runtime state value to be sure that the forthcoming event will not constitute a security policy violation, and then call the unpacked class's method if not. After the call, the wrapper method updates the runtime state value and repackages the class. If the call would have constituted a security policy violation, the object is repacked and a security exception is thrown. The strategy for testing runtime state values is the one described in §3.2.4 for testing whether a DFA is in a particular state.

Operations **pack** and **unpack** are implemented as method calls to the (very small) trusted C# library given in Figure 4.3. Observe that C#'s `lock` construct is used to make both operations atomic. This implementation suffices to prevent multithreaded programs from violating the security policy, but it will also cause some policy-satisfying programs to terminate prematurely. This can happen when two threads attempt to access the same package simultaneously. One thread will unpack the package first and the other will throw the `EmptyPackage` exception and terminate. To overcome this limitation, an alternative `Unpack` implementation could block when it encounters an empty package, waiting for the thread that

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

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

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

Figure 4.3: Implementation of **pack** and **unpack**

```
[Package]Package.Pack
        [o1 = [mscorlib]System.Object : h1]
        (_, C<o1>, H<[System]System.Object : h1>)
        [Ret([ ])]
[Package]Package.Unpack
        [ ]
        (_, &H<[System]System.Object : h1>)
        [ Ret([o1 = [System]System.Object : h1]C<o1>),
          Exn([ ]_) ]
```

Figure 4.4: Event declarations for `Pack` and `Unpack`

unpacked the package to repack it. We leave such an implementation to future work.

To permit use of our package implementation, a security policy must declare trusted signatures for our `Pack` and `Unpack` methods. It can do this by including the two event declarations given in Figure 4.4. The first declares that the `Pack` method takes an unpacked, security-relevant object and a runtime state value that represents its state, and causes the object to disappear from the typing context. Thus, any future references to it until it is unpacked are invalid and will be rejected by the type-checker. The second declares that the `Unpack` method takes a pointer to a location suitable for storing runtime state values and stores into it a value representing the state of the unpacked, security-relevant object it returns. If it

instead throws an exception, no security-relevant object is introduced to the typing context.

4.5 Experimental Results

To test the runtime overhead of our implementation, we applied our rewriter and type-checker to a managed C# port of the SciMark benchmark suite. We wrote a security policy that identifies as security-relevant events the various interesting method calls that the benchmark suite makes to the .NET runtime libraries. These include calls to the `Math` library, calls to the `StringBuilder` library, and calls to the `System.IO` library. Our security policy rejects programs that attempt more than n calls to these libraries, where n is a parameter set by the policy-writer. In order to prevent our policy from actually causing the benchmarks to be halted prematurely, we set n to -1, causing our rewriter to insert all the security checks that it would for any other value of n, but with tests that never signal a policy violation.

Our experiment consisted of three phases. First, we applied our rewriter to the benchmark suite and measured the runtime of the rewritten binary. Second, we hand-optimized the rewritten binary and inserted additional hand-written annotations to reflect what a better rewriter could achieve while still allowing the type-checker to certify that the code was policy-adherent. Third, we manually instrumented the original code with security checks that enforce the same security policy, but in a way that would not satisfy a Mobile type-checker. This is suggestive of the best runtime that can be achieved if security without certification is acceptable.

All three phases were performed on a 3.1GHz Pentium running version 1.0.3705 of the Microsoft .NET Framework. The unmodified SciMark benchmark suite consisted of 2000 lines of C# code and compiled to a CIL bytecode binary that was 32K in size. Applying our rewriter took 0.43 seconds and did not change the binary's size. (Padding introduced by the CLI binary format masked the small overhead introduced by additional instructions and annotations.) Hand-counting the material inserted by the rewriter revealed 3477 bytes in annotations and 725 bytes in additional instructions. Type-checking the rewritten binary took 0.50 seconds.

The average runtimes of our tests over one hundred trials are given in Table 4.1. Our rewriter produced runtime overheads of about 6% or less in most cases. In one case the overhead was almost a factor of 2. We expect that this would be the worst case for our rewriting strategy since it transforms security-relevant method calls into two method calls. When we hand-optimized the rewritten code, we were able to hoist some of the security checks out of the inner loops of the benchmarks, reducing that overhead to a more reasonable 3%. The uncertified, hand-instrumented code was slightly faster than what could be achieved with certified code. It yielded an average overhead of about 1%.

To test our rewriter and type-checker under a more realistic scenario, we next used our rewriter to enforce a security policy that allows each .NET network socket object to accept at most n connections during the program's lifetime (where n is a parameter specified by the policy-writer). Such a policy might be used, for example, to force applications to relinquish control of network ports after a certain amount of activity. We applied this policy to a small multithreaded webserver written in C# (about 600 lines of code). The original application binary was 20K

Table 4.1: SciMark benchmark runtimes

	original	rewritten	hand-optimized	uncertified
FFT	8.20s	8.61s (+5.00%)	8.52s (+3.90%)	8.27s (+0.85%)
LU Factorize	8.13s	15.88s (+95.33%)	8.38s (+3.08%)	8.30s (+2.09%)
Method Call	17.69s	18.42s (+4.13%)	18.40s (+4.01%)	18.03s (+1.92%)
StringBuilder	15.09s	15.34s (+1.66%)	15.20s (+1.66%)	15.17s (+0.53%)
File IO	10.02s	10.67s (+6.49%)	10.22s (+2.00%)	10.20s (+1.80%)
Total	59.13s	68.92s (+16.56%)	60.72s (+2.69%)	59.97s (+1.42%)

in size, and once again, rewriting did not alter its size. Hand-counting the material inserted by the rewriter revealed approximately 83 bytes in additional instructions and 117 bytes in annotations. Rewriting took 0.12 seconds and type-checking took 0.09 seconds on a 1.8GHz Pentium running version 1.0.3705 of the Microsoft .NET Framework. We benchmarked both the original and rewritten webservers by using WebStone to simulate two clients retrieving five webpages ranging in size from 500 bytes to 5 megabytes. WebStone reported that the rewritten webserver exhibited an average throughput rate that was 99.97% of the original webserver's.

4.6 Related Work

To our knowledge there has been no previous implementation of a certifying IRM system or of an IRM system for the .NET CLI; however, a variety of mechanisms have been developed to implement IRM's for Java and x86 programs without certification.

PoET (Policy Enforcement Toolkit) [ES99, Erl04] and SASI (Security Automata SFI Implementation) [ES00, Erl04] implement IRM's in Java and x86 programs from policies expressed as security automata. Security automata for PoET/SASI are deterministic, finite-state automata with transition relations labeled by predicates involving program operations (Java bytecode or x86 instructions) and their operands (e.g., method call parameters). In addition to the declarative components, security policy specifications also include operational components—policies can provide trusted code to be in-lined into the untrusted code. PoET and SASI have been used to enforce high-level policies, like the policy that prohibits Java programs from accessing the network after accessing the file system, as well as low-level policies like Java stack-inspection and x86 memory safety.

Naccio [ET99] enforces resource bound policies over Java and x86 programs. Policy enforcement is achieved by injecting code before and after each code point where a resource is accessed, similar to aspect weaving in an Aspect Oriented Programming framework [KLM+97]. The injected code is provided as part of the policy specification, making Naccio policies largely non-declarative. Since resource policies are usually per-object policies (i.e., they place a constraint on how many accesses are permitted for each object instance), Naccio includes strong support for per-object security policies.

Java-MaC (Monitoring and Checking) [KVK+04] is a system for runtime verification [BG05] that instruments Java programs according to policy specifications that consist of two components: a low-level component that defines primitive events and system states, and a high-level component that defines histories over those events and states. To provide strong formal guarantees, Java-MaC policies are strictly declarative and focus on warning the user of imminent policy violations rather than on taking direct corrective action automatically. The system does not support per-object policies and assumes each monitored object can be assigned a static name.

Java-MOP [CR03, CDR04, CR05] offers an Aspect-Oriented style framework for designing, developing, and implementing IRM's. It consists of a collection of various engines and plug-ins that can be mixed and matched to generate IRM code from policies expressed in various logics, and implement them for various Java architectures. IRM's generated by the framework are implemented in AspectJ [KHH+01].

The Polymer system [BLW05] focuses on enforcing security policies that are composeable. Policies in Polymer have runtime implementations that can dynam-

ically query one another to enforce a composite policy built up from the various sub-policy components. Like the implementation of Mobile presented in this chapter, Polymer regards method calls as security-relevant events and can enforce per-object security policies.

4.7 Conclusions and Future Work

Our implementation of Mobile for managed Microsoft .NET CIL expresses security policies as star-free ω-regular expressions. We verify such policies in the presence of exceptions, concurrency, finalizers, and non-termination, demonstrating that Mobile can be scaled to real type-safe, low-level languages. Preliminary experimental evidence suggests that this certified program-rewriting strategy can be used to enforce security policies with reasonable runtime and code bloat overhead, and yet provides strong guarantees of correctness.

However, much work remains to be done before our implementation can be used to enforce real security policies in practical settings. Our type-checker supports only a limited language of security policies by defining events as instance method calls. Future work should allow security policies to define any CIL instruction as a security-relevant event, and should support global events (rather than just per-object events) in the manner described in §3.1.2. This would allow it to type-check policies that involve combinations of static and instance method calls, or that constrain the number of instructions that a program can execute on any particular run.

The type-checker also only supports history module plug-ins that express event sequences as deterministic, finite-state security automata. Support for the other history module plug-ins mentioned in §3.2.4 would permit more efficient modeling

of resource bound policies and support more powerful security policies that cannot be modeled by finite-state automata.

The policy specification language used in our implementation has many shortcomings. It provides a compact and highly technical representation of the set of entities, events, and event sequences that comprise a history-based security policy, but is therefore very difficult for humans to read or reason about. Since the security policy specification is trusted, a specification language that is prone to human error can constitute a significant weakness in a security system. Future work should consider how to represent policy specifications in a more human-readable form, and in a way that allows a user interface to assist in policy-writing by checking policy specifications for obvious errors and inconsistencies. Work in cognitive linguistics and perceptual symbol systems [Tal83, Lan90, Lan95, Sim95, Bar99] might provide useful high-level guidance on how to design and evaluate better policy languages that refer to entities, events, and schemas that encode program behaviors.

Annotations in our implementation sometimes take the form of CIL instruction sequences in order to avoid a limitation of the ILX SDK libraries. This approach would not work in a setting where the instructions used to encode annotations could, themselves, constitute security-relevant events (e.g., if a policy considers all instructions to be security-relevant events). Future work should address this by using custom annotations with bytecode indexes to annotate code points, or by placing annotations in a separate file. Alternatively, support for custom annotations at the bytecode level would be a useful addition to the .NET binary format that would facilitate this and a great deal of other research.

The rewriter in our implementation takes the naïve approach of wrapping each security-relevant method call in a new method whose body includes guard in-

structions that detect and prevent policy violations. This strategy does not take advantage of many opportunities that the Mobile type system affords for optimizing security checks. A better rewriter could in-line the bodies of some of these wrapper methods to reduce the number of method calls, could hoist some security checks outside of loops when it is safe to do so, and could avoid some security checks entirely by observing that some security-relevant events can never constitute a violation of the security policy. Although the .NET JIT compiler already performs some of these optimizations, it is unlikely to optimize away most **pack** and **unpack** operations because compilers are typically conservative with regard to aliasing analyses, preserving most heap operations—especially in code that is synchronized. Development of rewriters that take advantage of the extra optimization opportunities provided by Mobile's linear type system is important for demonstrating the power and flexibility of the type system, and for achieving better runtime overheads.

Our support for concurrency in this implementation has the disadvantage that although our rewriter is sound in the sense that it produces policy-adherent code, it is incomplete in that it is unable to successfully preserve the behavior of many policy-adherent, multithreaded programs. Much of that incompleteness is a result of our simplistic implementation of packages, which causes an exception to be thrown whenever an empty package is unpacked. To support far more multithreaded programs, an alternative package implementation could block when an empty package is unpacked, waiting for the thread that unpacked it to repack it. This would allow multiple threads to simultaneously access the same package, but in a way that prevents multiple threads from simultaneously exhibiting security-relevant events on the same object. Future work should investigate rewrit-

ing strategies that leverage such a package implementation to correctly rewrite more multithreaded applications.

Our rewriter and type-checker are stand-alone applications that are executed from the command line. A real implementation of Mobile must integrate both applications into the load path of the system so that untrusted code cannot circumvent the security enforcement mechanism. Doing this poses significant challenges and interesting research questions. Production-level .NET architectures usually load and JIT-compile assemblies in stages rather than all at once. For example, an assembly's metadata might be loaded first, and then each method's body might be JIT-compiled on its first invocation. Some methods might be recompiled later to permit the runtime system to optimize based on profiling information collected after the first compilation. .NET architectures also cache precompiled binaries so that they can be reused without re-invoking the JIT compiler. Future work should consider how certifying program-rewriters can be added to these complex load paths without significantly increasing the average startup time for launching applications.

Chapter 5

Conclusions

This dissertation provides a three-fold argument for why automated program-rewriting constitutes a compelling and effective means of enforcing security policies over untrusted code.

First, Chapter 2 gave a formal definition of program-rewriting, and it formally characterized the class of security policies enforceable by program-rewriters. We proved the informal intuition that program-rewriters can implement any policy enforceable by an execution monitor by implementing the execution monitor as an in-lined reference monitor. Furthermore, we showed that program-rewriters can enforce policies that no execution monitor can enforce. These policies can only be enforced by program-rewriters that perform code transformations beyond those modeled by in-lined reference monitors. We observed that this space of additional security policies remains largely unexplored, suggesting the need for future research.

Second, Chapter 3 presented a design strategy called certified in-lined reference monitoring, which allows program-rewriters to be developed without significantly enlarging a system's trusted computing base. In a certified in-lined reference monitoring system, the program-rewriter itself need not be trusted because it produces well-typed target code that can be verified by a smaller type-checker that is trusted. Our type system, called Mobile, supports both global and per-object security policies, and it can be leveraged to enforce such policies in settings that include concurrency, exceptions, finalizers, and non-termination. This allows the development of large and complex program-rewriters that perform aggressive opti-

mizing and that enforce rich classes of security policies without contributing extra complexity to the trusted computing base.

Third, Chapter 4 demonstrated that it is feasible to implement certified program-rewriters for real architectures. We described a prototype implementation of Mobile for the Microsoft .NET Framework and used it to enforce security policies expressed as ω-regular expressions of events, where events are method calls. We used our implementation to enforce these security policies over applications written in managed C#. Our preliminary experimental evidence indicates that the rewriter, the type-checker, and the rewritten code all exhibit reasonable runtimes, and we recorded reasonable size overheads for annotations and code added to executables during the rewriting process.

This three-fold argument for the program-rewriting approach to security policy enforcement also identified numerous directions for future research.

Our theoretical work in Chapter 2 revealed that, unlike other known classes of enforceable security policies, the class of policies enforceable by program-rewriting does not correspond to any class of the arithmetic hierarchy. This raises interesting questions about how to relate this class of problems to known classes from complexity theory. Future work should also consider our model under different computability constraints, such as constraints that limit security enforcement mechanisms to smaller (e.g., polynomial) time and space computations.

The type system developed in Chapter 3 takes an effective but somewhat blunt approach to solving the problems of tracking object security states and tracking aliases of security-relevant objects. Specifically, we require security-relevant objects with aliases to be encapsulated at runtime into package objects with limited interfaces. To reduce the overhead of these additional runtime operations, fu-

ture work should investigate how more-powerful type systems, such as those that analyze dataflow rather than just control flow, might be used to support more optimization strategies used by in-lined reference monitors.

Finally, the implementation described in Chapter 4 is of a preliminary prototype that leaves much room for future development. The prototype supports policies that identify method calls as security-relevant events, but future implementations should support richer languages of events, such as those that identify any instruction as an event parameterized by the instruction's arguments. Future implementations should also take advantage of more opportunities provided by the Mobile type system for optimizing rewritten code. Such research could provide additional evidence that program-rewriting systems can be implemented for real systems in such a way that powerful classes of security policies can be enforced efficiently.

Appendix A

Program Machine Semantics

There are many equivalent ways to formalize TM's. We define them as 4-tuples:

$$M = (Q, \delta, q_0, b)$$

- Q is a finite set of *states*.
- δ is the TM's *transition relation*. (Since our TM's are deterministic, δ is a total function.) For each state in Q and each symbol that could be read from the work tape, δ dictates whether the PM halts (H), reads a symbol from the input tape and continues, or continues without reading a symbol from the input tape. If the TM continues without reading an input symbol, then δ specifies the new TM state, the symbol written to the work tape, and whether the work tape head moves left (-1) or right (1). Otherwise if an input symbol is read, it specifies all of the above (the new TM state, the symbol written to the work tape, and whether the work tape header moves left or right) for each possible input symbol seen. Thus δ has type[1]

$$\begin{aligned}\delta : Q \times \Gamma \to \big(&\{H\} \uplus \\ &(Q \times \Gamma \times \{-1, 1\}) \uplus \\ &(\Gamma \to (Q \times \Gamma \times \{-1, 1\}))\big)\end{aligned}$$

- $q_0 \in Q$ is the initial state of the TM.
- $b \in \Gamma$ is the *blank* symbol to which all cells of the work tape are initialized.

[1]Set operator $\uplus$ denotes disjoint union.

The *computation state of a TM* is defined as a 5-tuple:

$$\langle q, \sigma, i, \kappa, k \rangle$$

where $q \in Q$ is the current finite control state; $\sigma, \kappa \in \Gamma^\omega$ are the contents of the input and work tapes; and $i, k \geq 1$ are the positions of the input and work tape heads. Take TM M to be (Q, δ, q_0, b). When M is provided input σ, it begins in computation state $\langle q_0, \sigma, 1, b^\omega, 1 \rangle$. The TM computation state then changes according to the following small-step operational semantics:

$$\langle q, \sigma, i, \kappa, k \rangle \longrightarrow_{TM} \langle q, \sigma, i, \kappa, k \rangle$$
$$\text{if } \delta(q, \kappa[k]) = H.$$

$$\langle q, \sigma, i, \kappa, k \rangle \longrightarrow_{TM} \langle q',\ \sigma,\ i,\ \kappa[..k-1]\, s\, \kappa[k+1..],\ \max\{1, k+d\} \rangle$$
$$\text{if } \delta(q, \kappa[k]) = (q', s, d).$$

$$\langle q, \sigma, i, \kappa, k \rangle \longrightarrow_{TM} \langle q',\ \sigma,\ i+1,\ \kappa[..k-1]\, s\, \kappa[k+1..],\ \max\{1, k+d\} \rangle$$
$$\text{if } \delta(q, \kappa[k])(\sigma[i]) = (q', s, d).$$

PM's are defined exactly as TM's except that they carry additional information corresponding to the trace tape. The *computation state of a PM* is defined as a triple:

$$\big\langle \langle q, \sigma, i, \kappa, k \rangle, \tau, n \big\rangle$$

where $\langle q, \sigma, i, \kappa, k \rangle$ is the computation state of a TM, $\tau \in \Gamma^*$ is the contents of the trace tape up to the trace tape head, and $n \geq 0$ is a computational step counter. Initially, PM $M = (Q, \delta, q_0, b)$ when provided input σ begins in computation state $\langle S, \epsilon, 0 \rangle$ where S is the initial computation state of TM M for input σ and ϵ denotes the empty sequence. The PM computation state then changes according to the

following operational semantics:

$$\langle S, \tau, n\rangle \longrightarrow_{PM} \langle S',\ \tau\, T(M, \sigma, n+1),\ n+1\rangle$$

where $S \rightarrow_{TM} S'$ and $T : TM \times \Gamma^\omega \times \mathbb{N} \rightarrow \Gamma^*$ is a trace mapping satisfying the constraints on trace mappings given in §2.2.1. (A concrete example is given below.)

We illustrate by giving a concrete example of a PM. This requires first specifying a Turing Machine and then giving a suitable trace mapping. Let Γ_0 be $\{0, 1, \#\}$. Next define event set E_0 by

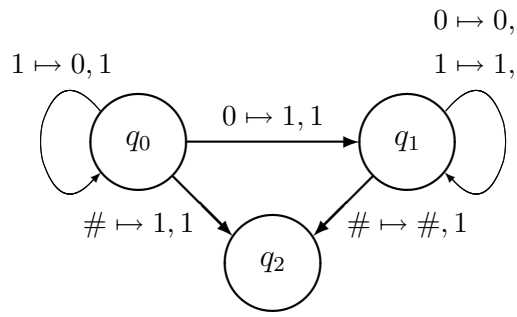
$$E_0 =_{\text{def}} \{e_s | s \in \Gamma_0\} \uplus \{e_{skip}, e_{end}\} \uplus \{e_M | M \in PM\}.$$

E_0 is a countable set, so there exists an unambiguous encoding of events from E_0 as finite sequences of symbols from Γ_0. Choose such an encoding and let $\lceil e \rceil$ denote the encoding of event $e \in E_0$. To avoid ambiguities in representing event sequences, choose the encoding so that for all $e \in E_0$, string $\lceil e \rceil$ consists only of symbols in $\{0, 1\}$ followed by a $\#$. This ensures that there exists a computable function $\lfloor \cdot \rfloor : \Gamma^\omega \rightarrow E^\omega$ such that for all $i \geq 0$ and for all $\chi \in E^i$, $\lfloor \lceil e_0 \rceil \cdots \lceil e_i \rceil \rfloor = e_0 \ldots e_i$.

Finally, for all $M \in TM$, $\sigma \in \Gamma^\omega$, and $n \geq 0$, define trace mapping T_0 by

$$\begin{aligned}
T_0(M, \sigma, 0) &=_{\text{def}} \lceil e_M \rceil. \\
T_0((Q, q_0, \delta, b), \sigma, n+1) &=_{\text{def}} \lceil e_{\sigma[i]} \rceil \quad \text{if } \langle q_0, \sigma, 1, b^\omega, 1\rangle \longrightarrow^n_{TM} \langle q, \sigma, i, \kappa, k\rangle, \text{and} \\
&\qquad\qquad \delta(q, \kappa[k]) \in \big(\Gamma \rightarrow (Q \times \Gamma \times \{-1, 1\})\big). \\
T_0((Q, q_0, \delta, b), \sigma, n+1) &=_{\text{def}} \lceil e_{end} \rceil \quad \text{if } \langle q_0, \sigma, 1, b^\omega, 1\rangle \longrightarrow^n_{TM} \langle q, \sigma, i, \kappa, k\rangle, \text{and} \\
&\qquad\qquad \delta(q, \kappa[k]) = H. \\
T_0(M, \sigma, n+1) &=_{\text{def}} \lceil e_{skip} \rceil \quad \text{otherwise.}
\end{aligned}$$

So, this trace mapping causes every PM M to write $\lceil e_M \rceil$ to its trace tape before its first computational step, write $\lceil e_s \rceil$ whenever it reads symbol s from its input tape,



We write $s \mapsto s', d$ (where $s, s' \in \Gamma$ and $d \in \{-1, 1\}$) by an arrow from state $q \in Q$ to $q' \in Q$ when reading s in state q causes the PM to write s' to its work tape, move the work tape header in direction d (i.e. left (-1) or right (1)), and transition to state q'.

$$M_{inc} =_{\text{def}} \big(\{q_0, q_1, q_2\}, \delta_{inc}, q_0, \#\big)$$

where for all $s \in \Gamma_0$,

$$\delta_{inc}(q_0, s) =_{\text{def}} \big(0 \mapsto (q_1, 1, 1);\ 1 \mapsto (q_0, 0, 1);\ \# \mapsto (q_2, 1, 1)\big)$$

$$\delta_{inc}(q_1, s) =_{\text{def}} \big(0 \mapsto (q_1, 0, 1);\ 1 \mapsto (q_1, 1, 1);\ \# \mapsto (q_2, \#, 1)\big)$$

$$\delta_{inc}(q_2, s) =_{\text{def}} \text{H}$$

Figure A.1: A PM for incrementing a counter

write $\lceil e_{skip} \rceil$ whenever it does not read an input symbol on a given computational step, and pad the remainder of the trace tape with $\lceil e_{end} \rceil$ if it halts.

For all $M \in PM$ and $\sigma \in \Gamma^\omega$, event sequence $\chi_{M(\sigma)}$ can be defined as

$$\chi_{M(\sigma)} =_{\text{def}} \lfloor \tau \rfloor$$

where τ is the limit as $n \to \infty$ of

$$\big\langle \langle q_0, \sigma, 1, b^\omega, 1\rangle, \epsilon, 0\big\rangle \longrightarrow_{TM}^{n} \big\langle \langle q, \sigma, i, \kappa, k\rangle, \tau, n\big\rangle$$

and $M = (Q, \delta, q_0, b)$. Therefore an enforcement mechanism could determine the sequence of events exhibited by a PM by observing the PM's trace tape.

Figure A.1 shows a program to increment binary numbers by 1, formalized as a PM along the lines we just discussed. The PM shown there treats its input as a two's-complement binary number (least-order bit first), and writes that number incremented by one to its work tape. As the PM executes, it also writes the sequence of symbols dictated by trace mapping T_0 to its trace tape. So if the PM

in Figure A.1 were provided string 1101 as input, it would write 0011 to its work tape and write $\lceil e_{M_{inc}} \rceil \lceil e_1 \rceil \lceil e_1 \rceil \lceil e_0 \rceil \lceil e_1 \rceil \lceil e_\# \rceil$ to its trace tape, followed by $\lceil e_{end} \rceil$ repeated through the remainder of the tape. A different PM M_0 that never reads its input would write to its trace tape $\lceil e_{M_0} \rceil$, then $\lceil e_{skip} \rceil$ for each computational step it takes, and finally $\lceil e_{end} \rceil$ repeated through the remainder of the tape.

We have given only one of many equivalent ways to formalize our program machines. Extra work tapes, multiple tape heads, multidimensional tapes, and two-way motion of the input tape head all yield computational models of equivalent power to the one we give. All of these models can simulate the operations found on typical computer systems, including arithmetic, stack-based control flow, and stack- and heap-based memory management. PM's can also simulate other PM's, which means they can perform the equivalent of runtime code generation. Program machines are thus an extremely flexible model of computation that can be used to simulate real computer architectures.

Appendix B

Proof of Type-soundness for Mobile

We here provide formal proofs of type-soundness and subject reduction for Mobile. These results are then used in Appendix C to prove the theorems of policy adherence and prefix adherence stated in §3.4.

B.1 Consistency of Statics and Dynamics

To formalize the theorems, we first provide a formal definition of the notion of consistency between static typing contexts and a runtime memory states described informally in §3.4. 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 inference rules given in Figure B.1. Rules B.2 – B.5 ensure that all values of object fields have the proper type. Rules B.6 – B.11 ensure that object traces are policy adherent and are adequately tracked by runtime history values in packages and by history maps when unpacked. Rules B.12 – B.13 ensure that items in the call stack are well-typed.

The proofs of Terminating Policy Adherence (Theorem 3.1) and of Non-terminating Prefix Adherence (Theorem 3.2) are arrived at in three steps. First, in §B.3 we prove subject reduction for the type system. That is, we prove that taking a step according to the operational semantics provided in Figure 3.6 preserves the type of a Mobile term as defined in Figure 3.7. Second, in §B.4 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

Figure B.1: Consistency of Mobile Statics and Dynamics

$$\frac{\Gamma \vdash_{heap} h : \Gamma \qquad \vdash_{hist} h : (\Gamma; \Psi) \qquad \Gamma \vdash_{stack} s : \overrightarrow{Fr}}{\Gamma \vdash (h, s) : (\Psi; \overrightarrow{Fr})} \tag{B.1}$$

$$\frac{\begin{array}{c}\Gamma_0 \vdash_{heap} h : \Gamma \\ \Gamma_0 \vdash \boxed{v_i} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; \mathit{field}(C, f_i))\ \forall i \in 1..\mathit{fields}(C)\end{array}}{\Gamma_0 \vdash_{heap} h, (\ell \mapsto \mathit{obj}_C\{f_i = v_i | i \in 1..\mathit{fields}(C)\}^{\overrightarrow{e}}) : \Gamma, \ell{:}C} \tag{B.2}$$

$$\frac{\Gamma_0 \vdash_{heap} h : \Gamma}{\Gamma_0 \vdash_{heap} h, (\ell \mapsto \mathit{pkg}(\ldots)) : \Gamma, \ell{:}C\langle ? \rangle} \tag{B.3}$$

$$\frac{\Gamma_0 \vdash_{heap} h : \Gamma}{\Gamma_0 \vdash_{heap} h : \Gamma, \theta} \tag{B.4}$$

$$\frac{}{\Gamma_0 \vdash_{heap} \cdot : \cdot} \tag{B.5}$$

$$\frac{\vdash_{hist} h : (\Gamma; \Psi) \qquad \overrightarrow{e} \subseteq H}{\vdash_{hist} h, (\ell \mapsto \mathit{obj}_C\{\ldots\}^{\overrightarrow{e}}) : (\Gamma, \ell{:}C; \Psi \star (\ell \mapsto H))} \tag{B.6}$$

$$\frac{\vdash_{hist} h : (\Gamma; \Psi) \qquad \overrightarrow{e} \subseteq H \subseteq \mathit{policy}(C)}{\begin{array}{r}\vdash_{hist} h, (\ell \mapsto \mathit{pkg}(\ell', \mathit{rep}_C(H))), (\ell' \mapsto \mathit{obj}_C\{\ldots\}^{\overrightarrow{e}}) : \\ (\Gamma, \ell{:}C\langle ? \rangle, \ell'{:}C; \Psi)\end{array}} \tag{B.7}$$

$$\frac{\vdash_{hist} h : (\Gamma; \Psi) \qquad \overrightarrow{e} \subseteq \mathit{policy}(C)}{\vdash_{hist} h, (\ell \mapsto \mathit{obj}_C\{\ldots\}^{\overrightarrow{e}}) : (\Gamma, \ell{:}C; \Psi)} \tag{B.8}$$

$$\frac{\vdash_{hist} h : (\Gamma; \Psi)}{\vdash_{hist} h, (\ell \mapsto \mathit{pkg}(\cdot)) : (\Gamma, \ell{:}C\langle ? \rangle; \Psi)} \tag{B.9}$$

$$\frac{\vdash_{hist} h : (\Gamma; \Psi)}{\vdash_{hist} h : (\Gamma, \theta; \Psi)} \tag{B.10}$$

$$\frac{}{\vdash_{hist} \cdot : (\cdot; 1)} \tag{B.11}$$

Figure B.1 (Continued)

$$\frac{\Gamma \vdash_{stack} s : \overrightarrow{Fr} \qquad \Gamma \vdash \boxed{v_i} : (\Psi; \overrightarrow{Fr}_0) \multimap (\Psi; \overrightarrow{Fr}_0; \tau_i) \ \forall i \in 0..n}{\Gamma \vdash_{stack} s(v_0, \ldots, v_n) : \overrightarrow{Fr}(\tau_0, \ldots, \tau_n)} \tag{B.12}$$

$$\frac{}{\Gamma \vdash_{stack} \cdot : \cdot} \tag{B.13}$$

an empty package. Third, these two results are leveraged in Appendix C to prove Terminating Policy Adherence and Non-terminating Prefix Adherence theorems.

B.2 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 B.1. (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) $h = h_0, (\ell \mapsto obj_C\{f_i = v_i | i \in 1..fields(C)\}^{\vec{e}})$

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

$$\frac{\Gamma' \vdash_{heap} h_0 : \Gamma_0 \qquad \Gamma' \vdash \boxed{v_i} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; field(C, f_i))\ \forall i \in 1..fields(C)}{\Gamma' \vdash_{heap} h : \Gamma} \text{(B.2)}$$

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

$$\frac{\Gamma' \vdash_{heap} h_0 : \Gamma_0}{\Gamma' \vdash_{heap} h : \Gamma} \text{(B.3)}$$

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 $\vdash_{hist} h : (\Gamma; \Psi)$ that ends in one of

$$\frac{\vdash_{hist} h_0 : (\Gamma_0; \Psi_0) \qquad \overrightarrow{e} \subseteq H}{\vdash_{hist} h : (\Gamma; \Psi)} (\text{B.6}) \,,$$

$$\frac{\vdash_{hist} h_1 : (\Gamma_1; \Psi) \qquad \overrightarrow{e} \subseteq H \subseteq \mathit{policy}(C)}{\vdash_{hist} h : (\Gamma; \Psi)} (\text{B.7}) \text{, or}$$

$$\frac{\vdash_{hist} h_0 : (\Gamma_0; \Psi) \qquad \overrightarrow{e} \subseteq \mathit{policy}(C)}{\vdash_{hist} h : (\Gamma; \Psi)} (\text{B.8})$$

where history map Ψ is defined by $\Psi = \Psi_0 \star (\ell' \mapsto H)$, typing context Γ_1 is defined by $\Gamma_1 = \Gamma_0, \ell{:}C\langle ? \rangle$, and heap h_1 is defined by $h_1 = h_0, (\ell \mapsto \mathit{pkg}(\ell', \mathit{rep}_C(H)))$;

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 $\vdash_{hist} h : (\Gamma; \Psi)$ that ends in one of

$$\frac{\vdash_{hist} h_1 : (\Gamma_1; \Psi) \qquad \overrightarrow{e} \subseteq H \subseteq \mathit{policy}(C)}{\vdash_{hist} h : (\Gamma; \Psi)} (\text{B.7})$$

or

$$\frac{\vdash_{hist} h_0 : (\Gamma_0; \Psi)}{\vdash_{hist} h : (\Gamma; \Psi)} (\text{B.9})$$

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

Facts 1 and 2 state that when $\Gamma' \vdash_{heap} h : \Gamma$ holds or $\vdash_{hist} h : (\Gamma; \Psi)$ 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 $\Gamma_0 \vdash_{heap} h : \Gamma$, judgment $\Gamma_0 \vdash_{stack} s : \overrightarrow{Fr}$, and judgment $\Gamma_0 \vdash I : (\Psi; \overrightarrow{Fr}) \multimap \exists \Gamma''.(\Psi''; \overrightarrow{Fr}''; \tau)$ can be weakened by adding more elements to Γ_0.

B.3 Subject Reduction

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

Proof. Observe that all typing rules except the typing rule for **ret** (3.41) 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 $\Gamma \vdash I : (\Psi; \overrightarrow{Fr}\, Fr) \multimap \exists \Gamma'.(\Psi'; \overrightarrow{Fr}\, Fr'; \tau)$ holds.

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

Case 1: $\mathcal{D}$ ends in Rule 3.19, 3.25, 3.30, 3.36, 3.37, 3.38, 3.39, or 3.40. In these cases, $\Psi' = \Psi$. The lemma follows immediately by instantiating Ψ with $\Psi_{extra} \star \Psi$ in each typing rule.

Case 2: $\mathcal{D}$ ends in Rule 3.20, 3.21, 3.22, 3.23, 3.24, 3.26, 3.27, 3.29, 3.31, 3.32, 3.33, or 3.34. The lemma follows by inductive hypothesis, by instantiating

each antecedent of the form $\Gamma_0 \vdash I_0 : (\Psi_0; Fr_0) \multimap \exists \Gamma_0'.(\Psi_0'; Fr_0'; \tau_0)$ with $\Gamma_0 \vdash I_0 : (\Psi_{extra} \star \Psi_0; Fr_0) \multimap \exists \Gamma_0'.(\Psi_{extra} \star \Psi_0'; Fr_0'; \tau_0)$.

Case 3: $\mathcal{D}$ ends in Rule 3.28. In addition to instantiating into each antecedent as in the previous case, instantiate Ψ_{unused} with $\Psi_{extra} \star \Psi_{unused}$. The lemma then holds by inductive hypothesis.

Case 4: $\mathcal{D}$ ends in Rule 3.35. Observe from the subtyping rules that if $\Psi_1 \preceq \Psi_1'$ then $\Psi_{extra} \star \Psi_1 \preceq \Psi_{extra} \star \Psi_1'$. We can therefore instantiate the rule's antecedents as in the previous two cases to prove the lemma by inductive hypothesis.

□

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

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

Base Case: If $\mathcal{D}$ ends with Rule B.11, then $\Psi = \Psi' = \cdot$ and the lemma holds immediately.

Inductive Case: If $\mathcal{D}$ ends in any remaining rule other than Rule B.6, then the lemma follows immediately from the inductive hypothesis. Assume $\mathcal{D}$ ends in Rule B.6 and therefore has the form

$$\frac{\vdash_{hist} h_0 : (\Gamma_0; \Psi_0) \qquad \overrightarrow{e} \subseteq H}{\vdash_{hist} h : (\Gamma; \Psi)} \text{(B.6)}$$

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

$$\frac{\vdash_{hist} h_0 : (\Gamma_0; \Psi_0') \qquad \overrightarrow{e} \subseteq H'}{\vdash_{hist} h : (\Gamma; \Psi')} \text{(B.6)}$$

□

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

$$\Gamma \vdash \psi : (\Psi; \overrightarrow{Fr}) \tag{B.14}$$

$$\Gamma \vdash I : (\Psi; \overrightarrow{Fr}) \multimap \exists \Gamma''.(\Psi''; \overrightarrow{Fr}''; \tau) \tag{B.15}$$

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 3.35 does not appear in the derivation of judgment B.15. Similar logic to that presented below applies to cases where Rule 3.35 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 3.19. To satisfy the lemma, choose $\Gamma' = \Gamma$, $\Psi' = \Psi$, $\overrightarrow{Fr}' = \overrightarrow{Fr}$, and $\sigma = \cdot$ and apply typing Rule 3.37.

Case 2: $\psi, E[I_0] \rightsquigarrow \psi', E[I_0']$. Let $\mathcal{D}$ be a derivation of B.15. 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: $\psi, \boxed{i4}\ I_2\ I_3\ \mathbf{cond} \rightsquigarrow \psi, I_j$ where $j \in \{2,3\}$. Any derivation of B.15 contains a subderivation of $\Gamma \vdash I_j : (\Psi; \overrightarrow{Fr}) \multimap \exists\Gamma''.(\Psi''; \overrightarrow{Fr}''; \tau)$ (by 3.20 and 3.37). 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 B.15 must have the form

$$\dfrac{\dfrac{\begin{array}{ccccc}\Gamma \vdash I_1 : (\Psi; \overrightarrow{Fr}) \multimap & & \Gamma \vdash I_2 : (\Psi_1; \overrightarrow{Fr}_1) \multimap & & \Gamma \vdash \boxed{\mathbf{0}} : (\Psi_1; \overrightarrow{Fr}_1) \multimap \\ \exists\Gamma_1.(\Psi_1; \overrightarrow{Fr}_1; \mathbf{int32}) & & \exists\Gamma_2.(\Psi; \overrightarrow{Fr}; \mathbf{void}) & & \exists\Gamma_2.(\Psi; \overrightarrow{Fr}; \mathbf{void})\end{array}}{\Gamma \vdash I_1\ I_2\ \boxed{\mathbf{0}}\ \mathbf{cond} : (\Psi; \overrightarrow{Fr}) \multimap \exists\Gamma''.(\Psi; \overrightarrow{Fr}; \mathbf{void})}(3.20)}{\Gamma \vdash I_1\ I_2\ \mathbf{while} : (\Psi; \overrightarrow{Fr}) \multimap \exists\Gamma''.(\Psi; \overrightarrow{Fr}; \mathbf{void})}(3.21)$$

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

$$\Gamma_0, \Gamma'' \vdash I_1\ (I_2; (I_1\ I_2\ \mathbf{while}))\ \boxed{\mathbf{0}}\ \mathbf{cond} : (\Psi; \overrightarrow{Fr}) \multimap \exists\Gamma''.(\Psi; \overrightarrow{Fr}; \mathbf{void})$$

using the typing derivation displayed in Figure B.2. 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 B.15 contains a subderivation of $\Gamma \vdash I_2 : (\Psi; \overrightarrow{Fr}) \multimap \exists\Gamma''.(\Psi''; \overrightarrow{Fr}''; \tau)$ (by 3.22 and 3.36). 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 B.15 and 3.25, $\overrightarrow{Fr}$ has the form $\overrightarrow{Fr}_0 Fr$ and $0 \leq j \leq n$. From B.14 and B.12, $Fr = (\tau_0, \ldots, \tau_n)$ 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 B.15 and 3.26, $\overrightarrow{Fr}$ has the form $\overrightarrow{Fr}_0 Fr$, and $0 \leq j \leq n$. From B.14 and B.12, $Fr = (\tau_0, \ldots, \tau_n)$. From

$$\dfrac{\dfrac{\Gamma \vdash I_1 : (\Psi; \overrightarrow{Fr}) \multimap \exists \Gamma_1.\ (\Psi_1; \overrightarrow{Fr}_1; \mathbf{int32})}{\Gamma \vdash I_1 : (\Psi; \overrightarrow{Fr}) \multimap \exists \Gamma''.\ (\Psi_1; \overrightarrow{Fr}_1; \mathbf{int32})}(3.35) \quad \dfrac{\dfrac{\Gamma \vdash I_2 : (\Psi_1; \overrightarrow{Fr}_1) \multimap \exists \Gamma_2.(\Psi; \overrightarrow{Fr}; \mathbf{void})}{\Gamma \vdash I_2 : (\Psi_1; \overrightarrow{Fr}_1) \multimap \exists \Gamma''.(\Psi; \overrightarrow{Fr}; \mathbf{void})}(3.35) \quad \Gamma \vdash I_1\ I_2\ \mathbf{while} : (\Psi; \overrightarrow{Fr}) \multimap \exists \Gamma''.(\Psi; \overrightarrow{Fr}; \mathbf{void})}{\Gamma \vdash I_2; (I_1\ I_2\ \mathbf{while}) : (\Psi_1; \overrightarrow{Fr}_1) \multimap \exists \Gamma''.(\Psi; \overrightarrow{Fr}; \mathbf{void})}(3.21) \quad \dfrac{\Gamma \vdash \boxed{\mathbf{0}} : (\Psi_1; \overrightarrow{Fr}_1) \multimap \exists \Gamma_2.(\Psi; \overrightarrow{Fr}; \mathbf{void})}{\Gamma \vdash \boxed{\mathbf{0}} : (\Psi_1; \overrightarrow{Fr}_1) \multimap \exists \Gamma''.(\Psi; \overrightarrow{Fr}; \mathbf{void})}(3.35)}{\Gamma_0, \Gamma'' \vdash I_1\ (I_2; (I_1\ I_2\ \mathbf{while}))\ \boxed{\mathbf{0}}\ \mathbf{cond} : (\Psi; \overrightarrow{Fr}) \multimap \exists \Gamma''.(\Psi; \overrightarrow{Fr}; \mathbf{void})}(3.20)$$

Figure B.2: Typing derivation for `while` loops

B.15 and 3.26, $\Gamma'' = \cdot$, $\Psi'' = \Psi$, $\overrightarrow{Fr}'' = \overrightarrow{Fr}_0(\tau_0, \ldots, \tau_{j-1}, \tau', \tau_{j+1}, \ldots, \tau_n)$, $\tau = \textbf{void}$, 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 B.12 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 3.36, satisfying the lemma.

Case 8: $(h, s), \boxed{v_1} \ \ldots \ \boxed{v_n}$ **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 B.15 and 3.27, $\Gamma'' = \ell{:}C$, $\Psi'' = \Psi \star (\ell \mapsto \epsilon)$, $\overrightarrow{Fr}'' = \overrightarrow{Fr}$, 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 B.14 one can derive

$$\frac{\Gamma' \vdash_{heap} h : \Gamma \qquad \Gamma' \vdash \boxed{v_i} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; \mathit{field}(C, f_i)) \ \forall i \in 1..n}{\Gamma' \vdash_{heap} h' : \Gamma'} \text{(B.2)}$$

and derive

$$\frac{\vdash_{hist} h : (\Gamma'; \Psi) \qquad \epsilon \subseteq \epsilon}{\vdash_{hist} h' : (\Gamma'; \Psi')} \text{(B.2)}$$

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 3.38 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 B.15 and 3.28, $\overrightarrow{Fr}'' = \overrightarrow{Fr}$, and there exists $(\Psi_{in}, (\tau_0, \ldots, \tau_n))$, Ψ_{out}, Ψ_{unused}, and Fr_{out} such that $\Psi = \Psi_{unused} \star \Psi_{in}$, $\Psi'' = \Psi_{unused} \star \Psi_{out}$,

$$\Gamma \vdash \boxed{v_i} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; \tau_i) \ \forall i \in 0..n \qquad \text{(B.16)}$$

and

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

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 (Lemma B.1), 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

$$\frac{\Gamma \vdash I_0 : (\Psi; \overrightarrow{Fr}(\tau_0, \ldots, \tau_n)) \multimap \exists \Gamma''.(\Psi''; \overrightarrow{Fr} Fr_{out}; \tau)}{\Gamma \vdash I_0 \ \mathbf{ret} : (\Psi; \overrightarrow{Fr}(\tau_0, \ldots, \tau_n)) \multimap \exists \Gamma''.(\Psi''; \overrightarrow{Fr}; \tau)} (3.41)$$

(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 B.14, and therefore one can derive

$$\frac{\Gamma \vdash_{stack} s : \overrightarrow{Fr} \qquad \Gamma \vdash \boxed{v_i} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; \tau_i) \ \forall i \in 0..n}{\Gamma' \vdash_{stack} sa : \overrightarrow{Fr}'} (B.12)$$

by B.16 and B.12.

Case 10: $(h, sa), \boxed{v}\ \mathbf{ret} \rightsquigarrow (h, s), \boxed{v}$. By B.15 and 3.41, $\Gamma'' = \cdot$, $\Psi = \Psi''$, 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 B.15 has a subderivation of $\Gamma \vdash \boxed{v} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; \tau)$.

Since the typing rules for value expressions are independent of frame, one can therefore derive $\Gamma \vdash \boxed{v} : (\Psi; \overrightarrow{Fr}') \multimap (\Psi''; \overrightarrow{Fr}''; \tau)$. Furthermore, one derivation of B.14 has a subderivation of

$$\frac{\Gamma \vdash_{stack} s : \overrightarrow{Fr}'' \qquad \vdots}{\Gamma \vdash_{stack} sa : \overrightarrow{Fr}'' Fr_0} \text{(B.12)}$$

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

Case 11: $(h, s), \boxed{\ell}$ **ldfld** μ $C{::}f \rightsquigarrow (h, s), \boxed{v}$ where $h(\ell) = obj_C\{\ldots, f = v, \ldots\}^{\overrightarrow{e}}$. By B.15 and 3.23, $\Gamma'' = \cdot$, $\Psi = \Psi''$, and $\overrightarrow{Fr} = \overrightarrow{Fr}''$. Choose $\Gamma' = \Gamma$, $\Psi' = \Psi$, $\overrightarrow{Fr}' = \overrightarrow{Fr}$, and $\sigma = \cdot$. Any derivation of B.15 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 B.14.

Case 12: $(h, s), \boxed{\ell}\ \boxed{v}$ **stfld** μ $C{::}f_j \rightsquigarrow (h', s), \boxed{\mathbf{0}}$ where $h' = h[\ell \mapsto obj_C[f_j \mapsto v]]$, and $1 \leq j \leq \mathit{fields}(C)$, and $h(\ell) = obj_C\{f_i = v_i | i \in 1..\mathit{fields}(C)\}^{\overrightarrow{e}}$. By B.15 and 3.24, $\Gamma'' = \cdot$, $\Psi = \Psi''$, $\overrightarrow{Fr} = \overrightarrow{Fr}''$, and $\tau = \mathbf{void}$. 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 3.36. Furthermore, since one derivation of B.14 has a subderivation of

$$\frac{\begin{array}{c}\Gamma \vdash_{heap} h_0 : \Gamma_0 \\ \Gamma \vdash \boxed{v_i} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; \mathit{field}(C, f_i))\ \forall i \in 1..\mathit{fields}(C)\end{array}}{\Gamma \vdash_{heap} h : \Gamma} \text{(B.2)}$$

where $\Gamma = \Gamma_0, \ell{:}C$ and $h = h_0, (\ell \mapsto obj_C\{\ldots\}^{\overrightarrow{e}})$, and since B.15 implies that all three of $\Gamma \vdash \boxed{v} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; \mu)$, $\mathit{field}(C, f_j) = \mu$, and $\ell \in \mathit{Dom}(\Gamma')$ hold, one can derive

$$\frac{\begin{array}{c}\Gamma \vdash_{heap} h_0 : \Gamma_0 \\ \Gamma \vdash \boxed{v} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; \mathit{field}(C, f_j)) \\ \Gamma \vdash \boxed{v_i} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; \mathit{field}(C, f_i))\ \forall i \in 1..j-1, j+1..\mathit{fields}(C)\end{array}}{\Gamma' \vdash_{heap} h' : \Gamma'} \text{(B.2)}$$

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

Case 13: $(h,s), \boxed{\ell}$ **evt** $e_1 \rightsquigarrow (h',s), \boxed{\mathbf{0}}$ where $h' = h[\ell \mapsto obj_C\{\ldots\}^{\overrightarrow{e}e_1}$ and $h(\ell) = obj_C\{\ldots\}^{\overrightarrow{e}}$. By B.15 and 3.29, $\Gamma'' = \cdot$, $\overrightarrow{Fr} = \overrightarrow{Fr}''$, $\tau = \mathbf{void}$, $\Psi = \Psi_1 \star (\ell \mapsto H)$ 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 3.36. Furthermore, since one derivation of B.14 has a subderivation of

$$\frac{\vdash_{hist} h_0 : (\Gamma_0; \Psi_1) \qquad \overrightarrow{e} \subseteq H}{\vdash_{hist} h : (\Gamma; \Psi)} \text{(B.6)}$$

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

$$\frac{\vdash_{hist} h_0 : (\Gamma_0; \Psi_1) \qquad \overrightarrow{e}e_1 \subseteq He_1}{\vdash_{hist} h' : (\Gamma; \Psi'')} \text{(B.6)}$$

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

Case 14: (h,s), **newpackage** $C \rightsquigarrow (h',s), \boxed{\ell}$ where $h' = h, \ell \mapsto pkg(\cdot)$. By B.15 and 3.30, $\Gamma'' = \ell{:}C\langle ? \rangle$, $\Psi = \Psi''$, $\overrightarrow{Fr} = \overrightarrow{Fr}''$, 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 3.39. Hence $\Gamma' \vdash \boxed{\ell} : (\Psi'; \overrightarrow{Fr}') \multimap \exists \Gamma''.(\Psi''; \overrightarrow{Fr}''; \tau)$ holds by Rule 3.35. In addition, any derivation of B.14 includes subderivations of $\Gamma \vdash_{heap} h : \Gamma$ and $\vdash_{hist} h : (\Gamma; \Psi)$; hence one can derive

$$\frac{\Gamma' \vdash_{heap} h : \Gamma}{\Gamma' \vdash_{heap} h' : \Gamma'} \text{(B.3)} \qquad \text{and} \qquad \frac{\vdash_{hist} h : (\Gamma; \Psi)}{\vdash_{hist} h' : (\Gamma'; \Psi')} \text{(B.9)}$$

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

Case 15: $\boxed{\ell}$ $\boxed{\ell'}$ $\boxed{\mathit{rep}_C(H)}$ **pack** $\rightsquigarrow (h',s), \boxed{\mathbf{0}}$ where $h(\ell) = pkg(\ldots)$ and $h' = h[\ell \mapsto pkg(\ell', \mathit{rep}_C(H))]$. By B.15 and 3.31, $\Gamma'' = \cdot$, $\overrightarrow{Fr} = \overrightarrow{Fr}''$, $\tau = \mathbf{void}$, and $\Psi = \Psi'' \star (\ell \mapsto H')$ for some H'. Any derivation of B.15 has subderivations of $\Gamma \vdash \boxed{\mathit{rep}_C(H)} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi''; \overrightarrow{Fr}; \mathcal{R}\mathit{ep}_C\langle H \rangle)$ (by Rule 3.40) such that

$H' \subseteq H \subseteq \mathit{policy}(C)$ (by Rule 3.31), and of

$$\frac{\Psi = \Psi'' \star (\ell' \mapsto H')}{\Gamma \vdash \boxed{\ell'} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; C\langle \ell \rangle)} \quad (3.38)$$

where $\Gamma = \Gamma_0, \ell{:}C\langle ? \rangle, \ell'{:}C$. One derivation of B.14 has a subderivation of

$$\frac{\dfrac{\mathcal{D}}{\vdash_{\mathit{hist}} h_0, (\ell \mapsto \mathit{pkg}(\ldots)) : (\Gamma_0, \ell{:}C\langle ? \rangle; \Psi'')} \qquad \overrightarrow{e} \subseteq H'}{\vdash_{\mathit{hist}} h : (\Gamma; \Psi)} \quad (B.6)$$

where $h = h_0, (\ell \mapsto \mathit{pkg}(\ldots)), (\ell' \mapsto \mathit{obj}_C\{\ldots\}^{\overrightarrow{e}})$ (because Rule B.6 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 3.36.

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

$$\frac{\vdash_{\mathit{hist}} h_0 : (\Gamma_0; \Psi'') \qquad \overrightarrow{e} \subseteq H \subseteq \mathit{policy}(C)}{\vdash_{\mathit{hist}} h' : (\Gamma; \Psi'')} \quad (B.7)$$

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

$$\mathcal{D} = \frac{\vdash_{\mathit{hist}} h_0 : (\Gamma_0; \Psi'')}{\vdash_{\mathit{hist}} h_0, (\ell \mapsto \mathit{pkg}(\cdot)) : (\Gamma_0, \ell{:}C\langle ? \rangle; \Psi'')} \quad (B.9)$$

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,

$$\mathcal{D} = \frac{\vdash_{\mathit{hist}} h_1 : (\Gamma_1; \Psi'') \qquad \overrightarrow{e''} \subseteq H'' \subseteq \mathit{policy}(C)}{\vdash_{\mathit{hist}} h_0, (\ell \mapsto \mathit{pkg}(\ell'', \mathit{rep}_C(H''))) : (\Gamma_0, \ell{:}C\langle ? \rangle; \Psi'')} \quad (B.7)$$

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

$$\frac{\vdash_{hist} h_1 : (\Gamma_1; \Psi'') \qquad \overrightarrow{e''} \subseteq policy(C)}{\vdash_{hist} h_0 : (\Gamma_0; \Psi'')} \text{(B.8)}$$

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

Case 16: $(h, s(v_0, \ldots, v_n)), \boxed{\ell}$ **unpack** $j \rightsquigarrow (h[\ell \mapsto pkg(\cdot)], sa'), \boxed{\ell'}$ where $h(\ell) = pkg(\ell', \mathit{rep}_C(H))$ and $a' = (v_0, \ldots, v_{j-1}, \mathit{rep}_C(H), v_{j+1}, \ldots, v_n)$. By B.15 and 3.34, $\Gamma'' = \ell{:}C, \theta$, $\Psi'' = \Psi \star (\ell \mapsto \theta)$, $\tau = C\langle\ell\rangle$, and

$$\overrightarrow{Fr}'' = \overrightarrow{Fr}_0(\tau_0, \ldots, \tau_{j-1}, \mathcal{R}ep_C\langle\theta\rangle, \tau_{j+1}, \ldots, \tau_n)$$

where $\overrightarrow{Fr} = \overrightarrow{Fr}_0(\tau_0, \ldots, \tau_n)$. One derivation of B.14 has a subderivation of

$$\frac{\vdash_{hist} h_0 : (\Gamma_0; \Psi) \qquad \overrightarrow{e} \subseteq H \subseteq policy(C)}{\vdash_{hist} h : (\Gamma; \Psi)} \text{(B.7)}$$

where $\Gamma = \Gamma_0, \ell{:}C\langle?\rangle, \ell'{:}C$ and

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

Choose $\Gamma' = \Gamma$, $\sigma = (\theta \mapsto \overrightarrow{e})$, $\Psi' = \sigma(\Psi'')$, and $\overrightarrow{Fr}' = \sigma(\overrightarrow{Fr}'')$. Since $\theta \notin Dom(\Gamma)$ (by Rule 3.32), it follows that $\sigma(\Psi'') = \Psi \star (\ell \mapsto \overrightarrow{e})$. One can therefore derive

$$\frac{\Psi' = \Psi \star (\ell \mapsto \overrightarrow{e})}{\Gamma' \vdash \boxed{\ell'} : (\Psi'; \overrightarrow{Fr}') \multimap (\sigma(\Psi''); \sigma(\overrightarrow{Fr}''); \sigma(\tau))} \text{(3.38)}$$

and one can derive

$$\frac{\dfrac{\vdash_{hist} h_0 : (\Gamma_0; \Psi)}{\vdash_{hist} h_0, (\ell \mapsto pkg(\cdot)) : (\Gamma_0, \ell{:}C\langle?\rangle; \Psi)} \text{(B.9)} \qquad \overrightarrow{e} \subseteq \overrightarrow{e}}{\vdash_{hist} h' : (\Gamma'; \Psi')} \text{(B.6)}$$

Finally, since any derivation of B.14 has a subderivation of

$$\frac{\Gamma \vdash_{stack} s : \overrightarrow{Fr}_0 \qquad \Gamma \vdash \boxed{v_i} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; \tau_i)\ \forall i \in 0..n}{\Gamma \vdash_{stack} s(v_0, \ldots, v_n) : \overrightarrow{Fr}_0(\tau_0, \ldots, \tau_n)} \text{(B.12)}$$

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 3.40, it follows from derivation Rule B.12 that $\Gamma' \vdash_{stack} sa' : \overrightarrow{Fr}'$ 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 $\Gamma' = \Gamma$, $\overrightarrow{Fr}'' = \overrightarrow{Fr}$,

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

and $\sigma = \cdot$. By B.15, 3.34, and 3.40, the typing judgments

$$\Gamma \vdash I_2 : (ctx^+_{C,k}; \overrightarrow{Fr}) \multimap \exists \Gamma''.(\Psi''; \overrightarrow{Fr}''; \tau)$$

and

$$\Gamma \vdash I_3 : (ctx^-_{C,k}; \overrightarrow{Fr}) \multimap \exists \Gamma''.(\Psi''; \overrightarrow{Fr}''; \tau)$$

both hold, so $\Gamma \vdash I_j : (\Psi'; \overrightarrow{Fr}') \multimap \exists \Gamma''.(\Psi''; \overrightarrow{Fr}''; \tau)$ holds.

Any derivation of B.14 has subderivations of $\Gamma \vdash_{heap} h : \Gamma$ and $\vdash_{hist} h : (\Gamma; \Psi)$. To prove that $\Gamma' \vdash \psi : (\Psi'; \overrightarrow{Fr}')$, it suffices to show that $\vdash_{hist} h : (\Gamma; \Psi')$. If $j = 2$ then $\mathrm{test}_{C,k}(\mathit{rep}_C(H)) \neq 0$ (by 3.17), and axiom 3.43 therefore implies that $\Psi \preceq ctx^+_{C,k}(H, \Psi)$. Alternatively, if $j = 3$ then $\mathrm{test}_{C,k}(\mathit{rep}_C(H)) = 0$, and axiom 3.42 therefore implies that $\Psi \preceq ctx^-_{C,k}(H, \Psi)$. In either case, $\Psi \preceq \Psi'$ holds. By context subtyping (Lemma B.2), we conclude that $\vdash_{hist} h : (\Gamma; \Psi')$ also holds.

Case 18: $\psi, \boxed{v_1}\ \ldots\ \boxed{v_n}$ **newhist** $C, k \rightsquigarrow \psi, \boxed{hc_{C,k}(v_1, \ldots, v_n)}$. By B.15 and 3.34, $\Gamma'' = \cdot$, $\Psi = \Psi''$, $\overrightarrow{Fr} = \overrightarrow{Fr}''$, and $\tau = HC_{C,k}(\mathcal{R}\mathit{ep}_{C_1}\langle H_1\rangle, \ldots, \mathcal{R}\mathit{ep}_{C_n}\langle H_n\rangle)$ where $\Gamma \vdash \boxed{v_i} : (\Psi; \overrightarrow{Fr}) \multimap (\Psi; \overrightarrow{Fr}; \mathcal{R}\mathit{ep}_{C_i}\langle H_i\rangle)$ holds for all $i \in 1..n$. Choose $\Gamma' = \Gamma$, $\Psi' = \Psi$, $\overrightarrow{Fr}' = \overrightarrow{Fr}$, and $\sigma = \cdot$. By axioms 3.44 and 3.45, there exists H such that $\tau = \mathcal{R}\mathit{ep}_C\langle H\rangle$ and $hc_{C,k}(v_1, \ldots, v_n) = \mathit{rep}_C(H)$. Thus, $\Gamma \vdash \boxed{\mathit{rep}_C(H)} : (\Psi'; \overrightarrow{Fr}') \multimap (\Psi''; \overrightarrow{Fr}''; \mathcal{R}\mathit{ep}_C\langle H\rangle)$ holds by typing Rule 3.40.

□

Theorem B.1 (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* $\psi, I \rightsquigarrow^n \psi', I'$ *holds then there exist* Γ', Ψ', $\overrightarrow{Fr}'$, *and* $\sigma : \theta \rightarrow \overrightarrow{e}$ *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 $\psi, I \rightsquigarrow^n \psi', I'$ holds, there exist ψ_1 and I_1 such that $\psi, I \rightsquigarrow^{n-1} \psi_1, I_1$ holds and such that $\psi_1, I_1 \rightsquigarrow \psi', I'$ 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 stepwise subject reduction (Lemma B.3).

□

B.4 Progress

Theorem B.2 (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 store* ψ' *and an instruction* I' *such that* $(h, s), I \rightsquigarrow \psi', I'$.
3. $I = E\left[\boxed{\ell}\ \textbf{unpack}\ j\right]$ *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: $I = E[\mathbf{ldc.i4}\ i4]$. 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 3.20, $v = i4$ 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 3.25, $\overrightarrow{Fr} = \overrightarrow{Fr}_0(\tau_0, \ldots, \tau_n)$ and $0 \leq j \leq n$. Since $\Gamma \vdash_{stack} s : \overrightarrow{Fr}$, it follows from derivation Rule B.12 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 3.26, $\overrightarrow{Fr} = \overrightarrow{Fr}_0(\tau_0, \ldots, \tau_n)$ and $0 \leq j \leq n$. Since $\Gamma \vdash_{stack} s : \overrightarrow{Fr}$, it follows from derivation Rule B.12 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 3.27 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: $I = E[\boxed{v_0}\ \ldots\ \boxed{v_n}\ \textbf{callvirt}\ C{::}m.Sig]$. By typing Rule 3.28, there exists I_0 such that $methodbody(C{::}m.Sig) = I_0$. Hence, condition 2 holds with $I' = E[I_0\ \textbf{ret}]$ and $\psi' = (h, s(v_0, \ldots, v_n))$.

Case 9: $I = E[\boxed{v}\ \textbf{ldfld}\ \mu\ C{::}f]$. By typing Rule 3.23, $v = \ell$ 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 B.2 that $h(\ell) = obj_C\{\ldots, f = v, \ldots\}^{\overrightarrow{e}}$ for some value v. Hence, condition 2 holds with $I = E[\boxed{v}]$ and $\psi' = (h, s)$.

Case 10: $I = E[\boxed{v}\ \boxed{v'}\ \textbf{stfld}\ \mu\ C{::}f]$. By typing Rule 3.24, $v = \ell$ 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 B.2 that $h(\ell) = obj_C\{\ldots, f = v, \ldots\}^{\overrightarrow{e}}$ for some value v. Hence, condition 2 holds with $I = E[\boxed{\mathbf{0}}]$ and $\psi' = (h[\ell \mapsto obj_C[f \mapsto v']], s)$.

Case 11: $I = E[\boxed{v}\ \textbf{evt}\ e_1]$. By typing Rule 3.29, $v = \ell$ 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 B.2 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 3.29, $v = \ell$ 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 $v'' = \mathit{rep}_C(H)$ for some history abstraction H. Since $\Gamma \vdash_{heap} h : \Gamma$ holds, it follows from derivation Rule B.3 that $h(\ell) = pkg(\ldots)$. Hence, condition 2 of the theorem holds with $I' = E[\boxed{\mathbf{0}}]$ and $\psi' = (h[\ell \mapsto pkg(\ell', \mathit{rep}_C(H))], s)$.

Case 14: $I = E[\boxed{v}\ \textbf{unpack}\ j]$. By typing Rule 3.32, $v = \ell$ 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 B.12 that $s = s_0(v_0, \ldots, v_n)$. Since $\Gamma \vdash_{heap} h : \Gamma$, it follows from derivation Rule B.3 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 3.33, $v = \mathit{rep}_C(H)$ for some class C and history abstraction H. Condition 2 therefore holds with

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

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

Case 16: $I = E[\boxed{v_1}\ \ldots\ \boxed{v_n}\ \textbf{newhist}\ k]$. By typing Rule 3.34, $arity(HC_k) = n$. By axiom 3.45, 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 $\psi' = (h, s)$.

□

Appendix C

Proof of Policy-adherence for Mobile Programs

We here prove the theorems of Terminating Policy Adherence (Theorem 3.1) and Non-terminating Prefix Adherence (Theorem 3.2) of Mobile programs using the results from Appendix B. The theorems are restated below for convenience.

Theorem (Terminating Policy Adherence)**.** *Assume that a Mobile program is well-typed, and that, as per Definition 3.2, 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* $\psi, methodbody(C_{main}{::}main.Sig) \rightsquigarrow^*(h', s'), \boxed{v}$ *holds, then* h' *is policy-adherent.*

Proof. By subject reduction (Theorem B.1), 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

$$\Gamma' \vdash (h'; s') : (\sigma(\Psi_{out}); \sigma(Fr_{out}); \sigma(\boxed{v})) \tag{C.1}$$

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 C.1 with a subderivation of

$$\frac{\vdots \qquad \overrightarrow{e} \subseteq \sigma(\Psi_{out}(\ell))}{\vdash_{hist} h' : (\Gamma'; \Psi')} \text{(B.6)}$$

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 C.1 with either a

subderivation of

$$\frac{\begin{array}{cc}\vdots & \overrightarrow{e} \subseteq \cdots \subseteq policy(C)\end{array}}{\vdash_{hist} h' : (\Gamma'; \Psi')} \text{(B.7)}$$

or a subderivation of

$$\frac{\begin{array}{cc}\vdots & \overrightarrow{e} \subseteq policy(C)\end{array}}{\vdash_{hist} h' : (\Gamma'; \Psi')} \text{(B.8)}$$

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

Theorem (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* $(h, s), I \rightsquigarrow^n (h', s'), I'$ *holds, then* h' *is prefix-adherent.*

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 small-step relations $(h, s), I \rightsquigarrow^{n-1} (h_1, s_1), I_1$ and $(h_1, s_1), I_1 \rightsquigarrow (h', s'), I'$ both hold. By inductive hypothesis, h_1 is prefix-adherent. By subject reduction (Theorem B.1), 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 3.27 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 3.29 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

$$\frac{\vdots \qquad \overrightarrow{e} \subseteq \Psi_1(\ell)}{\vdash_{hist} (h_1; s_1) : (\Gamma_1; \Psi_1)} \text{(B.6)}$$

Thus, $\overrightarrow{e}\, e_1 \subseteq pre(policy(C))$, 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.

□

Appendix D

Proof of Decidability of Subset Relations

The typing rules presented in Figure 3.7 require a type-checker to decide subset relations over the language of history abstractions given in Figure 3.2. History abstractions are star-free ω-regular expressions [Lad77, Tho79] with variables and intersection. In general, deciding subset for such a language is intractable, but not every history abstraction expression can appear in practice. Our implementation of Mobile decides subset for a sub-language of the language of history abstractions. We present this sub-language below, we argue that it captures most of the useful history abstractions that can appear in practice, and we prove that subset over this language can be reduced to subset over the language of regular expressions.

D.1 History Variables and Intersection

Intersection is introduced into a history abstraction during type-checking by typing rule 3.33 (the typing rule for **condst**). In our implementation, this typing rule substitutes an expression of the form $\theta \cap H$ for each occurrence of variable θ, where H is a closed history abstraction. Since intersection is introduced in no other way, this reduces the language of history abstractions of interest to the following sub-language of the language given in Figure 3.2:

$$H ::= \epsilon \mid e \mid H_1 H_2 \mid H_1 \cup H_2 \mid H^\omega \mid V$$

$$V ::= \theta \mid V \cap C$$

$$C ::= \epsilon \mid e \mid C_1 C_2 \mid C_1 \cup C_2 \mid C^\omega$$

Since our history abstractions are intended to model security automata, each closed history abstraction C introduced by the **condst** typing rule denotes the set of traces that can cause the automaton to enter a particular state. Since the automata are deterministic, for any pair C_1, C_2 of these abstractions, either $C_1 = C_2$ or $C_1 \cap C_2 = \emptyset$. Thus, we can conservatively approximate a history abstraction of the form $\theta \cap C_1 \cap C_2$ with an abstraction of the form $\theta \cap C_1$. The latter is guaranteed to be a superset of the former, and no IRM that models security policies using deterministic security automata will be affected by the conservative approximation.[1] This further reduces the language of history abstractions to

$$H ::= \epsilon \mid e \mid H_1 H_2 \mid H_1 \cup H_2 \mid H^\omega \mid V$$

$$V ::= \theta \cap C$$

$$C ::= \epsilon \mid e \mid C_1 C_2 \mid C_1 \cup C_2 \mid C^\omega$$

History variables are further constrained in where they can appear. No typing rule allows an open history abstraction to be appended to a closed history abstraction. History variables introduced in conditional branches and in loops are required to alpha-vary at join points for those conditionals and loops so that there is only ever one unique history variable per history abstraction. (This ensures that there are only a finite number of history variables in scope at any given control flow point.) Our implementation therefore only supports history abstractions of

[1] IRM's that do not model security policies using deterministic automata will be affected in that they will not be able to usefully nest dynamic state tests. That is, doing a second state test within a conditional branch of another state test will not cause the type-checker to infer the conjunction of the two tests; rather, the type-checker will conservatively infer typing refinements from only one of the tests, ignoring the other.

the form

$$H ::= (\theta \cap C_1)C_2 \mid C$$

$$C ::= \epsilon \mid e \mid C_1C_2 \mid C_1 \cup C_2 \mid C^\omega$$

D.2 Reduction to Regular Expression Subset

In this section we reduce the subset relation for the above language to subset over regular expressions.

Subset problems for the above language can appear in one of five possible forms:

1. $\forall\theta.\big((\theta \cap C_1)C_2 \subseteq (\theta \cap C_1')C_2'\big)$,
2. $\forall\theta_1, \theta_2.\big((\theta_1 \cap C_1)C_2 \subseteq (\theta_2 \cap C_1')C_2'\big)$ where $\theta_1 \neq \theta_2$,
3. $\forall\theta.\big((\theta \cap C_1)C_2 \subseteq C\big)$,
4. $\forall\theta.\big(C \subseteq (\theta \cap C_1)C_2\big)$, or
5. $C \subseteq C'$.

That is, there is either one history variable on both sides of the subset problem (1), a different history variable on each side (2), a single history variable on one side but none on the other (3 and 4), or no history variables at all (5). In Theorems D.1–D.4, we show that each of the first four forms can be reduced to the fifth form. Then in Theorem D.5 we reduce the fifth form to the subset problem for regular expressions.

Lemma D.1. *Every closed, non-empty ω-regular expression has a finite-length member.*

Proof. Let H be a closed, non-empty ω-regular expression. Proof is by induction on the structure of expression H. If H has the form ϵ or e, then the lemma follows immediately. If H has the form H_1H_2 or $H_1 \cup H_2$, then the lemma follows from the inductive hypothesis. If H has the form H_1^* or H_1^ω, then the lemma holds because $\epsilon \in H$. □

Theorem D.1. *The following two statements are equivalent:*

(i) $\forall\theta.\big((\theta \cap C_1)C_2 \subseteq (\theta \cap C_1')C_2'\big)$

(ii) $(C_1C_2 \subseteq \emptyset) \vee ((C_1 \subseteq C_1') \wedge (C_2 \subseteq C_2'))$

Proof. We begin by proving that (ii) implies (i). If $C_1C_2 \subseteq \emptyset$, then $C_1 \subseteq \emptyset$ or $C_2 \subseteq \emptyset$ holds. It follows that $(\theta \cap C_1)C_2 = \emptyset$ holds and the theorem is proved. Assume instead that C_1 and C_2 are both non-empty, and that $C_1 \subseteq C_1'$ and $C_2 \subseteq C_2'$ hold. Then $(\theta \cap C_1) \subseteq (\theta \cap C_1')$ holds, and hence $(\theta \cap C_1)C_2 \subseteq (\theta \cap C_1')C_2'$ holds.

It remains to show that (i) implies (ii). Assume that for all sets θ, $(\theta \cap C_1)C_2 \subseteq (\theta \cap C_1')C_2'$ holds. If $C_1C_2 \subseteq \emptyset$ then $C_1 \subseteq \emptyset$ or $C_2 \subseteq \emptyset$, in which case the theorem follows immediately. Assume instead that C_1C_2 is non-empty, and hence C_1 and C_2 are both non-empty. First, we prove that $C_1 \subseteq C_1'$ holds. Instantiate $\theta = C_1 - C_1'$ (where "$-$" denotes set difference). Then $(C_1 - C_1')C_2 \subseteq \emptyset$ holds. Since C_2 is non-empty by assumption, it follows that $C_1 - C_1' = \emptyset$ holds, and therefore $C_1 \subseteq C_1'$ holds. Second, we prove that $C_2 \subseteq C_2'$ holds. Since C_1 is non-empty, Lemma D.1 guarantees that there exists a finite member $s \in C_1$. Instantiate $\theta = s$. Then $sC_2 \subseteq (s \cap C_1')C_2'$ holds. Since C_2 is non-empty, sC_2 is also non-empty, and therefore $(s \cap C_1')$ and C_2' are non-empty. Since $(s \cap C_1')$ is non-empty, it follows that $(s \cap C_1') = s$. Therefore $sC_2 \subseteq sC_2'$ holds, and we conclude that $C_2 \subseteq C_2'$ holds. □

Theorem D.2. *The following two statements are equivalent:*

(i) $\forall\theta_1,\theta_2.\big((\theta_1 \cap C_1)C_2 \subseteq (\theta_2 \cap C_1')C_2'\big)$ *where* $\theta_1 \neq \theta_2$

(ii) $C_1C_2 \subseteq \emptyset$

Proof. To prove that (i) implies (ii), assume that for all sets θ_1 and θ_2, $(\theta_1 \cap C_1)C_2 \subseteq (\theta_2 \cap C_1')C_2'$ holds. Instantiate $\theta_1 = C_1$ and $\theta_2 = \emptyset$. It follows that $C_1C_2 \subseteq \emptyset$ holds.

To prove that (ii) implies (i), assume instead that $C_1C_2 \subseteq \emptyset$ holds, and hence $C_1 \subseteq \emptyset$ or $C_2 \subseteq \emptyset$ hold. Then $(\theta \cap C_1)C_2 = \emptyset$ holds, and the theorem follows immediately. □

Theorem D.3. *The following two statements are equivalent:*

(i) $\forall\theta.\big((\theta \cap C_1)C_2 \subseteq C\big)$

(ii) $C_1C_2 \subseteq C$

Proof. To prove that (i) implies (ii), assume that for all sets θ, $(\theta \cap C_1)C_2 \subseteq C$ holds. Instantiate $\theta = C_1$ and it follows that $C_1C_2 \subseteq C$ holds.

To prove that (ii) implies (i), assume instead that $C_1C_2 \subseteq C$ holds. Then for all sets θ, $(\theta \cap C_1)C_2 \subseteq C_1C_2 \subseteq C$ holds, proving the theorem. □

Theorem D.4. *The following two statements are equivalent:*

(i) $\forall\theta.\big(C \subseteq (\theta \cap C_1)C_2\big)$

(ii) $C \subseteq \emptyset$

Proof. To prove that (i) implies (ii), instantiate $\theta = \emptyset$ in (i), and (ii) follows immediately. To prove that (ii) implies (i), observe that $C \subseteq \emptyset \subseteq (\theta \cap C_1)C_2$ for any set θ. □

The above four proofs demonstrate that subset problems involving variables and intersection can all be reduced to three or fewer instances of subset problems over closed, star-free ω-regular expressions. We now show that the subset problem for closed, star-free ω-regular expressions can be reduced to the subset problem for regular expressions (expressions with Kleene star but not ω).

Lemma D.2. *Let C be an ω-regular expression (possibly with Kleene star), and define R to be the same expression but with all ω's replaced with Kleene stars. The set denoted by R is the set of finite-length members of the set denoted by C.*

Proof. Proof is by induction on the structure of expression C.

Case 1: $C = \emptyset$, $C = \epsilon$, or $C = e$. The lemma is immediate because C contains only finite-length sequences and $C = R$.

Case 2: $C = C_1C_2$ or $C = C_1 \cup C_2$. Hence $R = R_1R_2$ or $R = R_1 \cup R_2$, where R_1 and R_2 are C_1 and C_2 (respectively) with any ω's replaced by Kleene stars. By inductive hypothesis, R_1 is the set of finite members of C_1 and R_2 is the set of finite members of C_2. It follows that R_1R_2 is the set of finite members of C_1C_2 and $R_1 \cup R_2$ is the set of finite members of $C_1 \cup C_2$.

Case 3: $C = C_1^\omega$ or $C = C_1^*$. Hence $R = R_1^*$ where R_1 is C_1 with any ω's replaced by Kleene stars. The finite members of C are the concatenations of finitely many finite members of C_1. By inductive hypothesis, R_1 is the set of finite members of C_1. Thus, R_1^* is the set of finite-length members of C.

□

Büchi automata are non-deterministic finite state automata, possibly with ϵ-transitions, in which an accepting path is one which visits any final state infinitely

often. We define two classes of Büchi automata that are useful for accepting star-free ω-regular languages:

Definition D.1 (Star-free Büchi automata)**.** A Büchi automaton A is *star-free* if every cycle in A contains a final state.

Definition D.2 (∞-free Büchi automata)**.** A Büchi automaton A has a *finite acceptance state* q if q is a final state with an ϵ-transition to itself, and every path through A that accepts a finite-length sequence leads to q where it loops. A Büchi automaton A is *∞-free* if it has a finite acceptance state q and q is reachable from every cycle in A.

Lemma D.3. *Let C be a star-free ω-regular expression. There exists a star-free, ∞-free Büchi automaton that accepts the language denoted by C.*

Proof. We will prove by induction on the structure of C that there exists a star-free, ∞-free Büchi automaton A that accepts the language $C - \{\epsilon\}$. This suffices to prove the lemma since if $\epsilon \in C$, then A augmented with an ϵ-transition from its start state to its finite acceptance state constitutes a star-free, ∞-free Büchi automaton that accepts C.

Case 1: $C = \emptyset$ or $C = \epsilon$. Define A to be a Büchi automaton with two states: a non-final start state and a finite acceptance state. Automaton A is star-free and ∞-free, and since the finite acceptance state is not reachable from the start state, it accepts no sequences. Thus, A accepts $C - \{\epsilon\}$.

Case 2: $C = e$. Define automaton A as in case 1, but additionally add an edge labeled with e from the start state to the finite acceptance state. Automaton A is star-free and ∞-free, and it accepts the language $\{e\}$.

Case 3: $C = C_1 \cup C_2$. By inductive hypothesis, there exist star-free, ∞-free Büchi automata A_1 and A_2 such that A_1 accepts language $C_1 - \{\epsilon\}$ and A_2 accepts language $C_2 - \{\epsilon\}$. Define A to be a Büchi automaton that consists of a start state, a finite acceptance state, and copies of A_1 and A_2 in which the ϵ-self-transitions of their finite acceptance states have been removed. In addition, augment A with four extra ϵ-transitions: two from the start state of A to the start states of the copies of A_1 and A_2, and two from the finite acceptance states in the copies A_1 and A_2 to the finite acceptance state of A. By construction, A accepts language $(C_1 - \{\epsilon\}) \cup (C_2 - \{\epsilon\}) = (C_1 \cup C_2) - \{\epsilon\}$. Since A_1 and A_2 are star-free and ∞-free, it follows that A is star-free and ∞-free.

Case 4: $C = C_1 C_2$. By inductive hypothesis, there exist star-free, ∞-free Büchi automata A_1 and A_2 such that A_1 accepts language $C_1 - \{\epsilon\}$ and A_2 accepts language $C_2 - \{\epsilon\}$. If C_1 or C_2 are empty, then $C = \emptyset$ and the lemma follows from case 1. If $C_1 = \epsilon$ or $C_2 = \epsilon$ then the lemma follows immediately from the inductive hypothesis. Therefore, assume that $C_1 - \{\epsilon\}$ and $C_2 - \{\epsilon\}$ are non-empty, and that A_1 and A_2 therefore accept non-empty languages. Define A to be a Büchi automaton that consists of a start state, a finite acceptance state, and copies of A_1 and A_2 in which the ϵ-self-transitions of their finite acceptance states have been removed. In addition, augment A with three extra ϵ-transitions: one from the start state of A to the start state of the copy of A_1, one from the finite acceptance state of the copy of A_1 to the start state of the copy of A_2, and one from the finite acceptance state of the copy of A_2 to the finite acceptance state of A. Using A, construct two more automata: Define A' to be automaton A with an extra ϵ-transition

from its start state to the start state of the copy of A_2, and define A'' to be automaton A with an extra ϵ-transition from the start state of the copy of A_2 to the finite acceptance state of A.

If $\epsilon \notin C_1$ and $\epsilon \notin C_2$ then A is a star-free, ∞-free automaton that accepts language $C - \{\epsilon\}$. If $\epsilon \in C_1$ and $\epsilon \notin C_2$ then A' is a star-free, ∞-free automaton that accepts language $C - \{\epsilon\}$. If $\epsilon \notin C_1$ and $\epsilon \in C_2$ then A'' is a star-free, ∞-free automaton that accepts language $C - \{\epsilon\}$. Finally, if $\epsilon \in C_1$ and $\epsilon \in C_2$ then C is the union of the languages accepted by automata A' and A''. The lemma therefore follows from case 3.

Case 5: $C = C_1^\omega$. By inductive hypothesis, there exists a star-free, ∞-free Büchi automaton A_1 that accepts language $C_1 - \{\epsilon\}$. Define automaton A to be automaton A_1 with an ϵ-transition added from its finite acceptance state to its start state. Observe that A accepts language $C - \{\epsilon\}$. Automaton A is star-free and ∞-free because A_1 is star-free and ∞-free, and any cycles introduced by the extra ϵ-transition include the finite acceptance state.

□

Lemma D.4 (Pumping Lemma for star-free ω-regular expressions)**.** *Let C be a star-free ω-regular expression and let s be an infinite sequence. $s \in C$ if and only if there exists a finite-length sequence t and a partitioning of s into finite-length sequences $s_0 s_1 s_2 \cdots = s$ such that for all $j \geq 1$, s_j is non-empty and $s_0 (s_1)^{n_1} \cdots (s_j)^{n_j} t \in C$ for all $n_1, \ldots, n_j \geq 0$.*

Proof. We first prove the forward implication. Assume $s \in C$. By Lemma D.3, there exists a star-free, ∞-free Büchi automaton A that accepts the language denoted by C. Thus, A includes an (infinite-length) accepting path p for sequence

s. Since s is infinite but A includes only a finite number of states, some state q appears infinitely often in path p. Without loss of generality, assume that path p contains no cycles from q to itself that include only ϵ-transitions. (If p includes any such cycles, then removing all such cycles from path p yields another infinite-length path p' that reads s. Since any infinite-length path through a star-free automaton is an accepting path, p' is an accepting path for s, and we choose p to be path p' instead.) Define s_0 to be the sequence read along path p before it first visits state q. For each $j \geq 1$, define s_j to be the sequence read along path p between the jth and $j+1$st visits of q. Observe that s_j is non-empty because p does not include any ϵ-cycles from q to itself. Since state q lies within a cycle and since A is ∞-free, the finite acceptance state of A is reachable from q. Choose a finite path from q to the finite acceptance state of A and define t to be the (finite-length) sequence read along that path. Since for all $j \geq 1$, s_j is a sequence read along a path in A from q to itself, it follows that for all $n \geq 0$, $(s_j)^n$ is also a sequence read along a path in A from q to itself. Hence, we conclude that A has an accepting path for $s_0(s_1)^{n_1} \cdots (s_j)^{n_j} t$ for all $j \geq 1$ and for all $n_1, \ldots, n_j \geq 0$.

We next prove the inverse of the forward implication. Assume $s \notin C$. Let t be an arbitrary finite sequence and let $s_0 s_1 s_2 \cdots = s$ be an arbitrary partitioning of s into finite-length sequences, each of which is non-empty save possibly for s_0. We will prove that there exists $j \geq 1$ such that $s_0 s_1 \cdots s_j t \notin C$. By Lemma D.3, there exists a star-free Büchi automaton A that accepts the language denoted by C. Since $s \notin C$, automaton A includes no accepting path for sequence s. That is, either A has one or more non-accepting paths that read s, or A has no paths at all that read s. The former cannot be the case because any path that reads s is infinite in length, and every infinite path through a star-free automaton is an

accepting path. Since $s \notin C$, there is therefore no path through A that reads s. Hence, there is some finite prefix of s for which A has no path that reads it. It follows that there is some $j \geq 1$ such that $s_0 s_1 \cdots s_j$ is not a prefix of any sequence in C, and thus $s_0 s_1 \cdots s_j t \notin C$. □

Theorem D.5. *Let C_1 and C_2 be closed, star-free, ω-regular expressions, and define R_1 and R_2 to be the same expressions but with all ω's replaced by Kleene stars. Then $C_1 \subseteq C_2$ if and only if $R_1 \subseteq R_2$.*

Proof. We first prove the forward implication. Assume that $C_1 \subseteq C_2$. Thus, the set of finite-length sequences in C_1 is a subset of the set of finite-length sequences in C_2. By Lemma D.2, R_1 is the set of finite-length sequences in C_1 and R_2 is the set of finite-length sequences in C_2, so we conclude that $R_1 \subseteq R_2$.

We next prove the inverse implication. Assume $R_1 \subseteq R_2$, and let $s \in C_1$ be given. We will prove that $s \in C_2$. If s is finite, then Lemma D.2 implies that $s \in R_1$, and since $R_1 \subseteq R_2$, it follows that $s \in R_2$. Lemma D.2 therefore implies that $s \in C_2$. If s is infinite, Lemma D.4 implies that there exists a finite sequence t and a partitioning of s into finite sequences $s_0 s_1 s_2 \cdots = s$ with s_j non-empty for all $j \geq 1$, such that set S defined by

$$S = \left\{ s_0 (s_1)^{n_1} \cdots (s_j)^{n_j} t \mid j \geq 1,\ n_1, \ldots, n_j \geq 0 \right\}$$

satisfies $S \subseteq C_1$. Since all members of set S are finite, Lemma D.2 proves that $S \subseteq R_1$. Since $R_1 \subseteq R_2$, it follows that $S \subseteq R_2$. Lemma D.2 then implies that $S \subseteq C_2$. Finally, from Lemma D.4, $S \subseteq C_2$ implies that $s \in C_2$. □

The theorems presented in this subsection yield a simple algorithm for deciding subset over the sub-language of history abstractions defined in the previous subsection. That is, Theorems D.1–D.4 reduce the subset problem for history abstractions

with variables and intersection to three or fewer instances of the subset problem for history abstractions without variables or intersection. Then Theorem D.5 shows that subset for history abstractions without variables or intersection can be computed by changing all ω's into Kleene stars and deciding subset for the resulting regular expressions.

BIBLIOGRAPHY

[ABEL05] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity: Principles, implementations, and applications. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CSS)*, pages 340–353, Alexandria, Virginia, November 2005.

[ADS86] Bowen Alpern, Alan J. Demers, and Fred B. Schneider. Safety without stuttering. *Information Processing Letters (IPL)*, 23(4):177–180, November 1986.

[And72] James P. Anderson. Computer security technology planning study vols. I and III. Technical Report ESD-TR-73-51, HQ Electronic Systems Division: Hanscom AFB, MA, Fort Washington, Pennsylvania, October 1972.

[AS87] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[Bar99] Lawrence W. Barsalou. Perceptual symbol systems. *Behavioral and Brain Sciences*, 22:577–660, 1999.

[BDL06] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In *Proceedings of the 14th International Symposium on Formal Methods (FM)*, Hamilton, Ontario, Canada, August 2006.

[Ber04] Andrew Bernard. *Engineering Formal Security Policies for Proof-Carrying Code*. PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, April 2004.

[BG05] Howard Barringer and Yuri Gurevich, editors. *Proceedings of the 5th Workshop on Runtime Verification*, Edinburgh, Scotland, United Kingdom, July 2005.

[BL02] Andrew Bernard and Peter Lee. Temporal logic for Proof-Carrying Code. In *Proceedings of the 18th International Conference on Automated Deduction*, pages 31–46, Copenhagen, Denmark, July 2002.

[BLW05] Lujo Bauer, Jay Ligatti, and David Walker. Composing security policies with Polymer. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 305–314, Chicago, Illinois, June 2005.

[CCP04] Jean-Marc Champarnaud, Fabien Coulon, and Thomas Paranthoën. Compact and fast algorithms for regular expression search. *International Journal of Computer Mathematics (IJCM)*, 81(4):383–401, April 2004.

[CD02] David G. Clarke and Sophia Drossopoulou. Ownership, encapsulation and disjointness of type and effect. In *Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 292–310, Seattle, Washington, November 2002.

[CDR04] Feng Chen, Marcelo D'Amorim, and Grigore Roşu. A formal monitoring-based framework for software development and analysis. In *Proceedings of the 6th International Conference on Formal Engineering Methods (ICFEM)*, pages 357–373, Seattle, Washington, November 2004.

[CFS06] Sylvain Conchon, Jean-Christophe Filliâtre, and Julien Signoles. Designing a generic graph library using ML functors. Submitted for publication, May 2006.

[Cha01] Jean-Marc Champarnaud. Subset construction complexity for homogeneous automata, position automata and ZPC-structures. *Theoretical Computer Science*, 267:17–34, 2001.

[CNP01] David G. Clarke, James Noble, and John M. Potter. Simple ownership types for object containment. In *Proceedings of the 15th European Conference for Object-Oriented Programming (ECOOP)*, pages 53–76, Budapest, Hungary, June 2001.

[CR03] Feng Chen and Grigore Roşu. Towards Monitoring-Oriented Programming: A paradigm combining specification and implementation. In *Proceedings of the 3rd Workshop on Runtime Verification*, volume 89.2, pages 108–127, Boulder, Colorado, July 2003.

[CR05] Feng Chen and Grigore Roşu. Java-MOP: A Monitoring Oriented Programming environment for Java. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 546–550, Edinburgh, Scotland, United Kingdom, April 2005.

[DF01] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 59–69, Snowbird, Utah, June 2001.

[DF04a] Robert DeLine and Manuel Fähndrich. The Fugue protocol checker: Is your software baroque? Technical Report MSR-TR-2004-07, Microsoft Research, Redmond, Washington, January 2004.

[DF04b] Robert DeLine and Manuel Fähndrich. Typestates for objects. In *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP)*, pages 465–490, Oslo, Norway, June 2004.

[DG71] P. Deutsch and C. A. Grant. A flexible measurement tool for software systems. In *Information Processing 71, Proceedings of the IFIP Congress*, volume 1, pages 320–326, Ljubljana, Yugoslavia, August 1971.

[ECM02] ECMA. *ECMA-335: Common Language Infrastructure (CLI)*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, second edition, December 2002.

[Eme90] E. Allen Emerson. *Handbook of Theoretical Computer Science*, chapter on Temporal and Modal Logic, pages 995–1072. Elsevier and MIT Press, 1990.

[Erl04] Úlfar Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Cornell University, Ithaca, New York, January 2004.

[ES99] Úlfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the New Security Paradigms Workshop (NSPW)*, pages 87–95, Caledon Hills, Ontario, Canada, September 1999.

[ES00] Úlfar Erlingsson and Fred B. Schneider. IRM enforcement of Java stack inspection. In *Proceedings of the 21st IEEE Symposium on Security and Privacy*, pages 246–255, Oakland, California, May 2000.

[ET99] David Evans and Andrew Twynman. Flexible policy-directed code safety. In *Proceedings of the 20th IEEE Symposium on Security and Privacy*, pages 32–45, Oakland, California, May 1999.

[Fon04] Philip W. L. Fong. Access control by tracking shallow execution history. In *Proceedings of the 25th IEEE Symposium on Security and Privacy*, pages 43–55, Berkeley, California, May 2004.

[FTA02] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, Berlin, Germany, June 2002.

[Göd31] Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.

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

[GS01] Andrew D. Gordon and Don Syme. Typing a multi-language intermediate code. In *Proceedings of the 28th ACM Symposium on Principles*

of Programming Languages (POPL), pages 248–260, London, England, United Kingdom, January 2001.

[Ham98] Kevin W. Hamlen. Proof-Carrying Code for x86 architectures. Undergraduate honor's thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1998.

[HMS05] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. Certified in-lined reference monitoring on .NET. Technical Report TR-2005-2003, Cornell University, Ithaca, New York, November 2005.

[HMS06a] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. Certified in-lined reference monitoring on .NET. In *Proceedings of the 1st ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, volume 1, pages 7–15, Ottawa, Ontario, Canada, June 2006.

[HMS06b] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. Computability classes for enforcement mechanisms. *ACM Transactions On Programming Languages And Systems (TOPLAS)*, 28(1):175–205, January 2006.

[Hoa69] Charles A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM (CACM)*, 12(10):576–580, October 1969.

[HRU76] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *Communications of the ACM (CACM)*, 19(8):461–471, August 1976.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, Massachusetts, 1979.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)*, volume 2072, pages 327–355, Budapest, Hungary, June 2001.

[KKLS01] Moonjoo Kim, Sampath Kannan, Insup Lee, and Oleg Sokolsky. JavaMaC: a run-time assurance tool for Java programs. In *Proceedings of the 1st International Workshop on Runtime Verification (RV)*, pages 218–235, Paris, France, July 2001.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Medhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, volume 1241, pages 220–242, Jyväskylä, Finland, June 1997.

[KS01] Andrew Kennedy and Don Syme. The design and implementation of generics for the .NET Common Language Runtime. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, Snowbird, Utah, June 2001.

[KVK+04] Moonjoo Kim, Mahesh Viswanathan, Sampath Kannan, Insup Lee, and Oleg V. Sokolsky. Java-MaC: A run-time assurance approach for Java programs. *Formal Methods in System Design*, 24(2):129–155, March 2004.

[Lad77] Richard E. Ladner. Application of model theoretic games to descrete linear orders and finite automata. *Information and Control*, 33:281–303, 1977.

[Lam71] Butler W. Lampson. Protection. In *Proceedings of the 5th Symposium on Information Sciences and Systems*, pages 437–443, Princeton, New Jersey, March 1971.

[Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering (TSE)*, 2:125–143, March 1977.

[Lan90] Ronald W. Langacker. *Concept, Image, and Symbol: The Cognitive Basis of Grammar*. Mouton de Gruyter, Berlin, Germany, 1990.

[Lan95] Barbara Landau. Multiple geometric representations of objects in languages and language learners. In P. Bloom, M. A. Peterson, L. Nadel, and M. F. Garrett, editors, *Language And Space*, pages 317–364, Cambridge, Massachusetts, 1995. MIT Press.

[LBW05a] Jarred Ligatti, Lujo Bauer, and David Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1–2):2–16, February 2005.

[LBW05b] Jay Ligatti, Lujo Bauer, and David Walker. Enforcing non-safety security policies with program monitors. In *Proceedings of the 10th European Symposium on Research in Computer Security (ESORICS)*, pages 355–373, Milan, Italy, September 2005.

[Ler06] Xavier Leroy. Formal certification of a compiler back-end, or: Programming a compiler with a proof assistant. In *Proceedings of the 33rd ACM Symposium on Principles of Programming Languages (POPL)*, pages 42–54, Charleston, South Carolina, January 2006.

[Lig06] Jay Ligatti. *Policy Enforcement via Program Monitoring*. PhD thesis, Princeton University, Princeton, New Jersey, June 2006.

[LY99] Tim Lindholm and Frank Yellin. *The Java™ Virtual Machine Specification*. Addison-Wesley, second edition, 1999.

[MAF05] Greg Morrisett, Amal Ahmed, and Matthew Fluet. L^3: A linear language with locations. In *Proceedings of the 7th International Conference on Typed Lambda Calculi and Applications (TLCA)*, pages 293–307, Nara, Japan, April 2005.

[Mar89] Leo Marcus. The search for a unifying framework for computer security. *IEEE Cipher — Newsletter of the Technical Committee on Security and Privacy*, pages 55–63, June 1989.

[MCG99] Greg Morrisett, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(3):527–568, May 1999.

[MM06] Stephen McCamant and Greg Morrisett. Evaluating SFI for a CISC architecture. In *Proceedings of the 15th USENIX Security Symposium*, Vancouver, British Columbia, Canada, August 2006.

[Mye99] Andrew C. Myers. Practical mostly-static information flow control. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL)*, pages 228–241, San Antonio, Texas, January 1999.

[Nac97] Carey Nachenberg. Computer virus-antivirus coevolution. *Communications of the ACM (CACM)*, 40(1):46–51, January 1997.

[Nec97] George C. Necula. Proof-Carrying Code. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL)*, pages 106–119, Paris, France, January 1997.

[NL96] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI)*, pages 229–243, Seattle, Washington, October 1996.

[NL98] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 333–344, Montreal, Quebec, Canada, June 1998.

[ORY01] Peter O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Proceedings of the 15th Annual Conference of the European Association for Computer Science Logic (EACSL)*, pages 1–19, Paris, France, 2001. Springer-Verlag.

[Pap95] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, Reading, Massachusetts, 1995.

[RC91] Jonathan Rees and William Clinger. Revised report on the algorithmic language Scheme. *ACM Lisp Pointers*, 4(3):1–55, July 1991.

[RG05] Michael F. Ringenburg and Dan Grossman. Types for describing coordinated data structures. In *Proceedings of the 2nd ACM International Workshop on Types in Language Design and Implementation (TLDI)*, pages 25–36, Long Beach, California, January 2005.

[RVJV99] Bert Robben, Bart Vanhaute, Wouter Joosen, and Pierre Verbaeten. Non-functional policies. In *Proceedings of the 2nd International Conference on Meta-Level Architectures and Reflection*, volume 1616, pages 74–92, Saint-Malo, France, July 1999.

[Sch00] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(1):30–50, February 2000.

[Sim95] Raffaele Simone. Iconic aspects of syntax: A pragmatic approach. In Raffaele Simone, editor, *Iconicity in Language*, pages 153–169. John Benjamins, 1995.

[Sma97] Christopher Small. MiSFIT: A tool for constructing safe extensible C++ systems. In *Proceedings of the 3rd USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 175–184, Portland, Oregon, June 1997.

[SMH01] Fred B. Schneider, Greg Morrisett, and Robert Harper. A language-based approach to security. *Lecture Notes in Computer Science (LNCS)*, 2000:86–101, 2001.

[SS75] Jerry H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[SS04] Christian Skalka and Scott F. Smith. History effects and verification. In *Proceedings of the 2nd Asian Programming Languages Symposium (APLAS)*, pages 107–128, Taipei, Taiwan, November 2004.

[SWM00] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *Proceedings of the 9th European Symposium on Programming (ESOP)*, volume 1782, pages 366–381, Berlin, Germany, March 2000.

[Sym01] Don Syme. ILX: Extending the .NET Common IL for functional language interoperability. In Nick Benton and Andrew Kennedy, editors, *Proceedings of the 1st International Workshop on Multi-Language Infrastructure and Interoperability*, volume 59.1, Florence, Italy, September 2001.

[Tal83] Leonard Talmy. How language structures space. In H. L. Pick Jr. and L. P. Acredolo, editors, *Spatial Orientation: Theory, Research, and Application*, pages 225–282, New York, 1983. Plenum Press.

[Tho79] Wolfgang Thomas. Star-free regular sets of ω-sequences. *Information and Control*, 42:148–156, 1979.

[Tur36] Allen M. Turing. On computable numbers, with an application to the Entscheidungsproblem. In *Proceedings of the London Mathematical Society, series 2*, volume 42, pages 230–265, 1936.

[Vis00] Mahesh Viswanathan. *Foundations for the Run-time Analysis of Software Systems*. PhD thesis, University of Pennsylvania, December 2000.

[Wal00] David Walker. A type system for expressive security policies. In *Proceedings of the 27th ACM Symposium on Principles of Programming Languages (POPL)*, pages 254–267, Boston, Massachusetts, January 2000.

[War79] Willis H. Ware. Security controls for computer systems. Technical Report R-609-1, Rand Corporation, October 1979.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP)*, pages 203–216, Asheville, North Carolina, December 1993.