Policy Enforcement via Program Monitoring

Jarred Adam Ligatti

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
By the Department of
Computer Science

June, 2006

Abstract

One way to guarantee that software behaves securely is to monitor programs at run time and check that they dynamically adhere to constraints specified by a security policy. Whenever a program monitor detects that untrusted software is attempting to execute a dangerous action, it takes remedial steps to ensure that only safe code actually gets executed. This thesis considers the space of policies enforceable by monitoring the run-time behaviors of programs and develops a practical language for specifying monitors' policies.

In order to delineate the space of policies that monitors can enforce, we first have to define exactly what it means for a monitor to enforce a policy. We therefore begin by building a formal framework for analyzing policy enforcement; we precisely define policies, monitors, and enforcement. Having this framework allows us to consider the enforcement powers of program monitors and prove that they enforce an interesting set of policies that we define and call the infinite renewal properties. We show how, when given any reasonable infinite renewal property, to construct a program monitor that provably enforces that policy.

In practice, the security policies enforced by program monitors grow more complex both as the monitored software is given new capabilities and as policies are refined in response to attacks and user feedback. We propose dealing with policy complexity by organizing policies in such a way as to make them composeable, so that complex policies can be specified more simply as compositions of smaller subpolicy modules. We present a fully implemented language and system called Polymer that allows security engineers to specify and enforce composeable policies on Java applications. We also formalize the central workings of Polymer by defining an unambiguous semantics for our language.

Acknowledgments

This thesis would not exist without the help of many people. I am especially thankful to Lujo Bauer and David Walker for collaborating on much of the research presented here. Their insights have had a major impact on this research, and working with them has been an entirely enjoyable experience. David Walker has also been a wonderful graduate advisor, always happy to suggest, listen to, and improve on research ideas. He made graduate school an incredibly rewarding and painless experience for me.

In addition, I am grateful to Andrew Appel and Greg Morrisett for providing valuable feedback on how to improve this thesis. I am also indebted to Ed Felten, Kevin Hamlen, Greg Morrisett, and Fred Schneider for suggesting important revisions to early versions of this work.

This research was supported in part by ARDA grant NBCHC030106, DARPA award F30602-99-1-0519, NSF grants CCR-0238328 and CCR-0306313, Army Research Office grant DAAD19-02-1-0389, and a Sloan Fellowship.

Contents

Abstract iii

1 Introduction **1**

1.1 Related Work 4

1.1.1 Related Theoretical Efforts 4

1.1.2 Related Policy-specification-language Efforts 8

1.2 Contributions 11

2 Modeling Monitors as Security Automata **15**

2.1 Notation 15

2.2 Policies and Properties 16

2.3 Security Automata 18

2.4 Property Enforcement 20

3 Policies Enforceable by Monitors **23**

3.1 Truncation Automata 23

3.1.1 Definition 24

3.1.2 Enforceable Properties 25

3.2 Edit Automata 32

3.2.1 Definition 32

3.2.2 Enforceable Properties 33
3.3 Infinite Renewal Properties 44
3.3.1 Renewal, Safety, and Liveness 44
3.3.2 Example Properties 45

4 Enforcing Policies with Polymer 50
4.1 Polymer's Approach to Policy Complexity 50
4.2 Polymer System Overview 52
4.3 The Polymer Language 54
4.3.1 Core Concepts 54
4.3.2 Simple Policies 59
4.3.3 Policy Combinators 64
4.4 Empirical Evaluation 72
4.4.1 Implementation 72
4.4.2 Case Study: Securing Email Clients 74
4.4.3 Specifying Non-safety Policies in Polymer 78

5 Formal Semantics of the Polymer Language 84
5.1 Syntax 85
5.2 Static Semantics 87
5.3 Dynamic Semantics 91
5.4 Semantics-based Observations 95
5.5 Type Safety 96

6 Conclusions 108
6.1 Summary 108
6.2 Future Work 111

6.3 Closing Remarks 114

Bibliography **115**

List of Figures

3.1 Relationships between safety, liveness, and renewal properties. . . . 48

4.1 A secure Polymer application 53

4.2 Basic Polymer API for `Action` objects 54

4.3 Polymer's abstract `Sug` class 58

4.4 The parent class of all Polymer policies 58

4.5 Polymer policy that allows all actions 59

4.6 Polymer policy that disallows `Runtime.exec` methods 60

4.7 Abstract action for receiving email messages; the action's signature is `Message[ ] GetMail()` 62

4.8 Abbreviated Polymer policy that logs all incoming email and prepends the string "SPAM:" to subject lines on messages flagged by a spam filter 63

4.9 Polymer policy that seeks confirmation before creating `.exe`, `.vbs`, `.hta`, and `.mdb` files 65

4.10 Lattice ordering of Polymer suggestions' semantic impact 67

4.11 A conjunctive policy combinator 69

4.12 The `TryWith` policy combinator 70

4.13 Email policy hierarchy 75
4.14 Polymer policy that only allows network connections to email ports 77
4.15 Non-safety Polymer policy ensuring that ATM cash dispensation gets logged properly 80
4.16 Abbreviated non-safety Polymer policy ensuring that files are eventually written satisfactorily 82

5.1 Formal syntax for the Polymer calculus 86
5.2 Static semantics (rules for policies, suggestions, actions, and programs) 89
5.3 Static semantics (rules for case expressions) 90
5.4 Static semantics (standard rules) 91
5.5 Evaluation contexts 92
5.6 Dynamic semantics (policy and target steps; beta steps for functions) 93
5.7 Dynamic semantics (beta steps for case expressions) 94
5.8 Dynamic semantics (standard beta steps) 95

Chapter 1

Introduction

A ubiquitous technique for enforcing software security is to dynamically monitor the behavior of programs and take remedial action when the programs behave in a way that violates a security policy. Firewalls, virtual machines, and operating systems all act as *program monitors* to enforce security policies in this way. We can even think of any application containing security code that dynamically checks input values, queries network configurations, raises exceptions, warns the user of potential consequences of opening a file, etc., as containing a program monitor *inlined* into the application. This thesis examines the space of policies enforceable by program monitors and develops a practical language for specifying monitors' policies.

Monitor-enforceable Policies Because program monitors, which react to the potential security violations of *target programs*, enjoy such ubiquity, it is important to understand their capabilities as policy enforcers. Such an understanding is essential for developing systems that support program monitoring and for developing sound languages for specifying the security policies that these systems can enforce. In addition, well-defined boundaries on the enforcement powers of security mecha-

nisms allow security architects to determine exactly when certain mechanisms are needed and save the architects from attempting to enforce policies with insufficiently strong mechanisms.

Schneider defined the first formal models of program monitors and discovered one particularly useful boundary on their power [48]. He defined a class of monitors that respond to potential security violations by halting the target application, and he showed that these monitors can only enforce *safety* properties—security policies that specify that "nothing bad ever happens" in a valid run of the target [35]. When a monitor in this class detects a potential security violation (i.e., "something bad"), it must halt the target.

Aside from our work, other research has likewise only focused on the ability of program monitors to enforce safety properties. In this thesis, we advance the theoretical understanding of practical program monitors by proving that certain types of monitors can enforce non-safety properties. These monitors are modeled by *edit automata*, which have the power to insert actions on behalf of, and suppress actions attempted by, the target application. We prove an interesting lower bound on the properties enforceable by such monitors: a lower bound that encompasses strictly more than safety properties.

A Policy-specification Language Unfortunately, the run-time policies we need to enforce in practice tend to grow ever more complex. The increased complexity occurs for several reasons. First, as software becomes more sophisticated, so do our notions of what constitutes valid and invalid behavior. Witness, for example, the increased complexity of reasoning about security in a multi-user and networked system versus a single-user, stand-alone machine. Security concerns grow even more

complex when we consider more sophisticated systems that support, for example, electronic commerce or medical databases.

Practical security policies also grow more complex as security engineers tighten policies in response to new attacks. When engineers discover an attack, they often add rules to their security policies (increasing policy complexity) to avoid the newly observed attacks. For instance, a security engineer might add policy rules that disallow insecure default configurations or that require displaying a warning and asking for user confirmation before downloading dangerous files.

Even when security engineers relax overly tight policies, their policies often become more complex. In this case, the complexity increases because the original policy forbade too much and needs more sophisticated reasoning to distinguish between safe and dangerous behaviors. For example, an older version of the Java Development Kit (JDK 1.0) required all applets to be sandboxed. User feedback led to the adoption of a more relaxed policy, that only unsigned applets be sandboxed, in a later version (JDK 1.1) [41]. This relaxation increased policy complexity by requiring the policy to reason about cryptographic signatures.

This thesis attacks the problem of policy complexity by developing a programming language in which complex policies can be specified more simply as compositions of smaller *subpolicy* modules. Our compositional design allows security architects to reuse, update, and analyze isolated subpolicies. We present a fully implemented language, called Polymer, for specifying and enforcing complex and composeable run-time policies on Java applications. We provide several examples of Polymer policies and define a formal semantics for an idealized subset of the language that contains all of the key features.

1.1 Related Work

Only a handful of efforts have been made to understand the space of policies enforceable by monitoring software at run time; in contrast, a rich variety of policy-specification languages has been implemented. This lack of theoretical work makes it difficult to understand exactly which sorts of security policies to expect implemented systems to be able to enforce. We next examine closely related theoretical and policy-specification-language efforts and discuss high-level similarities and differences between our work and the related projects. In the remainder of this thesis, we point out additional, more specific relationships between our results and those of related work.

1.1.1 Related Theoretical Efforts

Monitors As Invalid Execution Recognizers Schneider began the effort to understand the space of policies that monitors can enforce [48]. Building on earlier work with Alpern, which provided logic-based and automata-theoretic definitions of safety and liveness [6, 5], Schneider modeled program monitors as infinite-state automata using a particular variety of Büchi automata [15] (which are like regular deterministic finite automata except that they can have an infinite number of states, operate on infinite-length input strings, and accept inputs that cause the automaton to enter accepting states infinitely often). Schneider's monitors[1] observe executions of untrusted target applications and dynamically recognize invalid behaviors. When a monitor recognizes an invalid execution, it halts the target just before the execution becomes invalid, thereby guaranteeing the validity of all mon-

[1]Schneider refers to his models as *security automata*. In this thesis, we call them *truncation automata* and use the term security automata to refer more generally to any dynamic execution transformer. Section 2.3 presents our precise definition of security automata.

itored executions. Schneider formally defined policies and properties and observed that his automata-based execution recognizers can only enforce safety properties (a monitor can only halt the target upon observing an irremediably "bad thing").

This thesis builds on Schneider's definitions and models but develops a different view of program monitors as execution *transformers* rather than execution *recognizers*. This fundamental shift permits modeling the realistic possibility that a monitor might *insert* actions on behalf of, and *suppress* actions of, untrusted target applications. In our model, Schneider's monitors are *truncation automata*, which can only *accept* the actions of untrusted targets and *halt* the target altogether upon recognizing a safety violation. We define more general monitors modeled by *edit automata* that can insert and suppress actions (and are therefore operationally similar to deterministic I/O automata [40]), and we prove that edit automata are strictly more powerful than truncation automata (Section 3.2.2). We demonstrate concrete, practical monitors that enforce non-safety properties, and even pure liveness properties, in Section 4.4.3.

Computability Constraints on Execution Recognizers After Schneider showed that the safety properties constitute an upper bound on the set of policies enforceable by simple monitors, Viswanathan, Kim, and others tightened this bound by placing explicit computability constraints on the safety properties being enforced [51, 32]. Their key insight was that because execution recognizers inherently have to decide whether target executions are invalid, these monitors can only enforce decidable safety properties. Introducing computability constraints allowed them to show that monitors based on recognizing invalid executions (i.e., our truncation automata) enforce exactly the set of computable safety properties. Moreover,

Viswanathan proved that the set of languages containing strings that satisfy a computable safety property equals the set of coRE languages [51].

Shallow-history Execution Recognizers Continuing the analysis of monitors acting as execution recognizers, Fong defines *shallow history automata* (SHA) as a specific type of memory-bounded monitor [23]. SHA decide whether to accept an action by examining a finite and unordered history of previously accepted actions. Although SHA are very limited models of finite-state truncation automata, Fong interestingly shows that they can nonetheless enforce a wide range of useful access-control properties, including Chinese Wall policies (where subjects may access at most one element from every set of conflicting data [14]), low-water-mark policies (where a lattice of trustworthiness determines whether accesses are valid [13]), and one-out-of-k authorization policies (where every program has a predetermined, finite set of access permissions [17]). In addition, Fong generalizes SHA by defining sets of properties accepted by arbitrarily memory-bounded monitors and proves that classes of monitors with strictly more memory can enforce strictly more properties.

Fong simplifies his analyses by assuming that monitors observe only finite executions (i.e., all untrusted targets must eventually halt) and ignoring computability constraints on monitors. Although we do not make those simplifying assumptions in this thesis, we did when first exploring the capabilities of edit automata [9, 38].

Comparison of Enforcement Mechanisms' Capabilities Hamlen, Morrisett, and Schneider observe that in practice, program monitors are often implemented by *rewriting* untrusted target code [24]. A rewriter *inlines* a monitor's code directly into the target at compile or load time. Many of the implemented systems discussed

in the next subsection, and indeed our own Polymer system, operate in this way; Section 4.2 contains more details about this implementation technique.

Hamlen et al. define the set of *RW-enforceable policies* as the policies enforceable by rewriting untrusted target applications, and they compare this set with the sets of policies enforceable by static analysis and monitoring mechanisms. Their model of program monitors differs from ours in that their monitors have access to the full text (e.g., source code or binaries) of monitored target programs. Practical monitors often adhere to this assumption: operating systems and virtual machines can usually access the full code of target programs. However, practical monitors also often violate this assumption: firewalls, network scanners, and monitors that can only "hook" their code into security-relevant methods of an operating system API (such as the "cloaking" monitors installed by some DRM mechanisms [47]) lack access to target programs' code.

Hamlen et al. model programs as program machines (PMs), which are three-tape deterministic Turing Machines (one tape contains input actions, one is a work tape, and one tape contains output actions). They show that the set of statically enforceable properties on PMs equals the set of decidable properties of programs (which contains only very limited properties such as "the program halts within one hundred computational steps when the input is 1010"). Because their monitors have access to the code of target programs, their monitors can perform "static" analysis on PMs and hence enforce strictly more policies than can be enforced through static analysis alone. For example, one can monitor a program to ensure that it never executes a particular action, but this same property cannot be enforced by static analysis on general PMs. Hamlen et al. also show that the RW-enforceable policies are a superset of the monitor-enforceable policies and interestingly prove

that the RW-enforceable policies do not correspond to *any* complexity class in the arithmetic hierarchy.

1.1.2 Related Policy-specification-language Efforts

Implemented Policy-specification Languages Although relatively little theoretical work has been done to understand the policies enforceable by monitoring software, a rich variety of general policy-specification languages has been implemented [36, 28, 17, 16, 20, 19, 22, 21, 46, 33, 10, 18, 49, 25]. These systems provide languages in which security engineers can write a centralized policy specification; the systems then use a tool to automatically insert code into untrusted target applications (i.e., they *instrument* the target application) in order to enforce the centrally specified policy on the target application. This centralized-policy architecture makes reasoning about policies a simpler and more modular task than the alternative approach of scattering security checks throughout application or execution-environment code. With a centralized policy, it is easy to locate the policy-relevant code and analyze or update it in isolation. Our implemented policy-specification language and enforcement system, Polymer, also applies this centralized-policy architecture; Section 4.2 contains further details.

The implementation efforts cited above thus deal with one part of the policy complexity problem—they ensure that policies exist in a centralized location rather than being dispersed throughout application or execution-environment code. The next step in dealing with the problem of policy complexity is to break complex, though centralized, policies into smaller pieces. Although some of the cited projects support limited policy decomposition via fixed sets of policy *combinators* (higher-order policies that compose arbitrary subpolicies) [21, 28, 17, 16, 10], they lack

mechanisms to define new combinators that can arbitrarily modify previously written policies and dynamically create policies. We provide these abilities in Polymer by making policies first-class objects; arbitrary higher-order policies may be parameterized by and return other policies. The ability to compose policies in arbitrary ways allows us to write expressive combinators that flexibly reuse subpolicies. We explore some possibilities in Section 4.3.3.

In addition, none of the previous projects provide a methodology for making all policies composeable. Problems arise when we try to compose general policies without constraining their effects. For example, one policy module may print an error message and then halt the target when it observes what it considers to be an illegal target action a, while another policy might print an acceptance message and allow the target to continue operating when it observes a. If we enforce the conjunction of these two policies, we expect the target to be halted when observing a, but what, if anything, should be printed? Polymer distinguishes itself from the related work cited above by providing a methodology for dealing with these kinds of conflicting effects when policies are composed. Our solution is to separate policies into effectless *query* methods, which tell combinators how a policy reacts to security-sensitive actions, and effectful bookkeeping methods, which perform I/O and policy state updates. Chapter 4 explains this methodology in greater detail.

Semantics of Policy-specification Languages Of the implemented languages cited above, PoET/Pslang [20, 18] and Naccio [22] are the most closely related to Polymer because they support the specification of arbitrary imperative policies that contain both security state and methods to update security state when policy-relevant methods execute. A major difference between Polymer and these

closely related projects, in addition to the differences noted above regarding policy composeability, is that Polymer provides a precise, formal semantics for its core language (Chapter 5). We consider the semantics an important contribution because it distills and unambiguously communicates the central workings of the Polymer language.

Very recently, Krishnan created a monitoring policy calculus based on the semantics of Polymer [34]. He achieves simplicity by removing our compositionality constraint on policies (that they all be separated into effectless query methods and effectful bookkeeping methods). Krishnan encodes most of our policy combinators into his calculus, gives our combinators formal semantics, states interesting properties about the combinators such as associativity, and explains how to encode several types of policies, including dynamically updateable policies, email policies similar to the one we have implemented in Polymer (Section 4.4.2), and privacy policies, in his calculus. Our own earlier work also provides precise semantics for some policy combinators, but in a less general and more complicated semantics [10]. We showed how policies composed using a common, though fixed, set of combinators can be analyzed statically to ensure that their effects do not conflict dynamically.

Aspect-oriented Languages Our policy-specification language can also be viewed as an aspect-oriented programming language (AOPL) [31] in the style of AspectJ [30]. The main high-level differences between our work and previous AOPLs are that our "aspects" (the program monitors) are first-class values and that we provide mechanisms to allow programmers to explicitly control the composition of aspects. Several researchers [50, 53] describe functional, as opposed to object-oriented, AOPLs with first-class aspect-oriented advice and formal semantics. However, they do not

support aspect combinators like the ones we develop here. In general, composing aspects is a known problem for AOPLs, and we hope the ideas presented here will suggest a new design strategy for general-purpose AOPLs.

1.2 Contributions

This thesis extends previous work in four principal ways.

1. Beginning with standard definitions of policies and properties, we introduce formal models of program monitors and define precisely how these monitors enforce policies by *transforming* possibly nonterminating target executions (Chapter 2). We consider this formal framework a central contribution of our work because it not only communicates our basic assumptions about what constitutes a policy, a monitor, and enforcement of a policy by a monitor, but also enables rigorous analyses of monitors' enforcement capabilities.

2. We use our formal framework to delineate the space of policies enforceable by two varieties of run-time program monitors: simple *truncation automata* and more sophisticated *edit automata* (Chapter 3). We also define an interesting set of security policies called the *infinite renewal properties*, and show how, when given any reasonable infinite renewal property, to construct a program monitor that provably enforces that policy. Infinite renewal properties include some non-safety properties, and we demonstrate how example monitors enforce non-safety properties.

3. We describe Polymer, a language for specifying complex run-time security policies more simply as compositions of smaller subpolicy modules (Chapter 4).

We make policies composeable by incorporating two primary innovations into our language.

- We separate policies into effectless methods that generate *suggestions* about how to deal with security-relevant application events and effectful methods that update security state only under certain conditions. This organization allows policy writers to deal with the possibly conflicting effects of composed policies.
- We make policies, suggestions, and application events first-class objects, so that higher-order policies can *query* subpolicy objects for suggestions about how to handle application events and combine those suggestions in meaningful ways.

We develop a library of common policy combinators and use them to build a complex security policy for untrusted email clients. In addition, we demonstrate concrete, practical examples of Polymer monitors that enforce non-safety (in fact, purely liveness) properties. Our language, libraries, and example policies are fully implemented and available for download from the Polymer project website [12].

4. We formalize semantics for an idealized version of our language that includes all of the key features of our implementation (Chapter 5). The formal semantics helps nail down corner cases and provides an unambiguous specification of how security policies execute. We prove that our language is type safe, a necessary property for protecting our program monitors from untrusted applications.

Most of these contributions were first presented in a series of workshop, journal, and conference papers written in collaboration with Lujo Bauer and David Walker [9, 38, 11, 39]. Significant portions of this thesis describing theoretical definitions of policies, monitors, and enforcement, as well as the properties enforceable by program monitors, come from a recent paper entitled "Enforcing Non-safety Security Policies with Program Monitors" [39], while the Polymer material chiefly comes from "Composing Security Policies with Polymer" [11]. This thesis extends the results of those papers in many ways.

- We include proofs for the theorems in Chapter 3. These theorems delineate upper and lower bounds for the properties enforceable by various monitoring mechanisms. Most importantly, the proof of Theorem 8 shows how, when given any reasonable infinite renewal property, to construct a program monitor that provably enforces that property.

- We provide significantly more complete code examples in Chapter 4 to fully illustrate the key elements of designing and implementing Polymer policies.

- We demonstrate in Section 4.4.3 two practical examples of non-safety Polymer policies. This ties the theoretical analysis in Section 3.2, which proves that monitors can enforce some non-safety policies, to our language for specifying monitor policies in practice.

- We include the complete semantics of the Polymer language in Chapter 5. This semantics precisely specifies the full meaning of the idealized version of our language.

- We also include the proof of type safety for the formal Polymer language in Chapter 5. Although our proof techniques are standard, having a proof formally assures us that our language is sound.

- Finally, we make numerous minor corrections and additions to the original material. For instance, we have fixed a bug in our notation for sequence concatenation[2] in Section 2.1 and added some high-level information to the discussion of policy combinators in Section 4.3.3.

[2]The original notation only applied when concatenating two finite sequences, but we often have to notate the concatenation of finite and infinite sequences to denote a finite sequence followed by an infinite sequence.

Chapter 2

Modeling Monitors as Security Automata

This chapter sets up a formal framework for analyzing policies, monitors, and enforcement. Chapter 3 uses this framework in its formal analysis of the policies that can be enforced by monitoring software.

We begin in Section 2.1 by describing some basic notation for specifying program executions. Then, Section 2.2 defines policies and properties, and Section 2.3 defines program monitors as security automata. Finally, Section 2.4 links together the previous definitions in order to define precisely what it means for a monitor to enforce a policy.

2.1 Notation

We specify a system at a high level of abstraction as a nonempty, possibly countably infinite set of *program actions* $\mathcal{A}$ (also referred to as program events). An *execution* is simply a finite or infinite sequence of actions. The set of all finite executions on a

system with action set $\mathcal{A}$ is notated as $\mathcal{A}^\star$. Similarly, the set of infinite executions is $\mathcal{A}^\omega$, and the set of all executions (finite and infinite) is $\mathcal{A}^\infty$. We let the metavariable a range over actions, σ and τ over executions, and Σ over sets of executions (i.e., subsets of $\mathcal{A}^\infty$).

The symbol $\cdot$ denotes the empty sequence, that is, an execution with no actions. We use the notation $\tau;\sigma$ to denote the concatenation of two sequences, the first of which must have finite length. When τ is a (finite) prefix of (possibly infinite) σ, we write $\tau \preceq \sigma$ or, equivalently, $\sigma \succeq \tau$. Given some σ, we often use $\forall \tau \preceq \sigma$ as an abbreviation for $\forall \tau \in \mathcal{A}^\star : \tau \preceq \sigma$; similarly, when given some τ, we abbreviate $\forall \sigma \in \mathcal{A}^\infty : \sigma \succeq \tau$ simply as $\forall \sigma \succeq \tau$.

2.2 Policies and Properties

A *security policy* is a predicate P on sets of executions; a set of executions $\Sigma \subseteq \mathcal{A}^\infty$ satisfies a policy P if and only if $P(\Sigma)$. For example, a set of executions satisfies a nontermination policy if and only if every execution in the set is an infinite sequence of actions. A cryptographic key-uniformity policy might be satisfied only by sets of executions such that the keys used in all the executions form a uniform distribution over the universe of key values.

Following Schneider [48], we distinguish between *properties* and more general policies as follows. A security policy P is a *property* if and only if there exists a *characteristic predicate* $\hat{P}$ over $\mathcal{A}^\infty$ such that for all $\Sigma \subseteq \mathcal{A}^\infty$, the following is true.

$$P(\Sigma) \iff \forall \sigma \in \Sigma : \hat{P}(\sigma) \qquad \text{(Property)}$$

Hence, a property is defined exclusively in terms of individual executions and may not specify a relationship between different executions of the program. The nontermination policy mentioned above is therefore a property, while the key-uniformity policy is not. The distinction between properties and policies is an important one to make when reasoning about program monitors in our current framework because a monitor only sees individual executions and can therefore enforce only security properties rather than more general policies.

There is a one-to-one correspondence between a property P and its characteristic predicate $\hat{P}$, so we use the notation $\hat{P}$ unambiguously to refer both to a characteristic predicate and the property it induces. When $\hat{P}(\sigma)$, we say that σ *satisfies* or *obeys* the property, or that σ is *valid* or *legal*. Likewise, when $\neg\hat{P}(\tau)$, we say that τ *violates* or *disobeys* the property, or that τ is *invalid* or *illegal*.

Properties that specify that "nothing bad ever happens" are called *safety properties* [35, 6]. No prefix of a valid execution can violate a safety property; stated equivalently: once some finite execution violates the property, all extensions of that execution violate the property. Technically, safety means that every invalid execution has some invalid prefix after which all extensions are likewise invalid. Formally, $\hat{P}$ is a safety property on a system with action set $\mathcal{A}$ if and only if the following is true.[1]

$$\forall\sigma \in \mathcal{A}^\infty : (\neg\hat{P}(\sigma) \implies \exists\sigma' \preceq \sigma : \forall\tau \succeq \sigma' : \neg\hat{P}(\tau)) \qquad \text{(Safety)}$$

Many interesting security policies, such as access-control policies, are safety properties, since security violations cannot be "undone" by extending a violating execution.

[1] Alpern and Schneider [6] model executions as infinite-length sequences of states, where terminating executions contain a final state, infinitely repeated. We can map an execution in their model to one in ours simply by sequencing the events that induce the state transitions (no event induces a repeated final state). With this mapping, it is easy to verify that our definitions of safety and liveness are equivalent to those of Alpern and Schneider.

Dually to safety properties, *liveness properties* [6] state that nothing exceptionally (i.e., irremediably) bad can happen in any finite amount of time. Any finite sequence of actions can always be extended so that it satisfies the property. Formally, $\hat{P}$ is a liveness property on a system with action set $\mathcal{A}$ if and only if the following is true.

$$\forall \sigma \in \mathcal{A}^\star : \exists \tau \succeq \sigma : \hat{P}(\tau) \qquad \text{(Liveness)}$$

The nontermination policy is a liveness property because any finite execution can be made to satisfy the policy simply by extending it to an infinite execution.

General properties may allow executions to alternate freely between satisfying and violating the property. Such properties are neither safety nor liveness but instead a combination of a single safety and a single liveness property [5]. We show in Chapter 3 that edit automata effectively enforce an interesting new sort of property that is neither safety nor liveness.

2.3 Security Automata

Program monitors operate by *transforming* execution sequences of an untrusted target application at run time to ensure that all observable executions satisfy some property. We model a program monitor formally by a *security automaton* S, which is a deterministic finite or countably infinite state machine (Q, q_0, δ) that is defined with respect to some system with action set $\mathcal{A}$. The set Q specifies the possible automaton states, and q_0 is the initial state. Different automata have slightly different sorts of transition functions (δ), which accounts for the variations in their expressive power. The exact specification of a transition function δ is part of the definition of each kind of security automaton; we only require that δ be complete,

deterministic, and Turing Machine computable. We limit our analysis in this work to automata whose transition functions take the current state and input action (the next action the target wants to execute) and return a new state and at most one action to output (make observable). The current input action may or may not be consumed while making a transition.

We specify the execution of each different kind of security automaton S using a labeled operational semantics. The basic single-step judgment has the form $(q, \sigma) \xrightarrow{\tau}_S (q', \sigma')$ where q denotes the current state of the automaton, σ denotes the sequence of actions that the target program wants to execute, q' and σ' denote the state and action sequence after the automaton takes a single step, and τ denotes the sequence of at most one action output by the automaton in this step. The input sequence, σ, is not observable to the outside world whereas the output, τ, is observable.

We generalize the single-step judgment to a multi-step judgment using standard rules of reflexivity and transitivity.

Definition 1 (Multi-step)

The multi-step relation $(\sigma, q) \overset{\tau}{\Longrightarrow}_S (\sigma', q')$ *is inductively defined as follows (where all metavariables are universally quantified).*

1. $(q, \sigma) \overset{\cdot}{\Longrightarrow}_S (q, \sigma)$
2. *If* $(q, \sigma) \xrightarrow{\tau_1}_S (q'', \sigma'')$ *and* $(q'', \sigma'') \overset{\tau_2}{\Longrightarrow}_S (q', \sigma')$ *then* $(q, \sigma) \overset{\tau_1;\tau_2}{\Longrightarrow}_S (q', \sigma')$

In addition, we define what it means for a program monitor to *transform* a possibly infinite-length input execution into a possibly infinite-length output execution. This definition is essential for understanding the behavior of monitors operating on potentially nonterminating targets.

Definition 2 (Transforms)

A security automaton $S = (Q, q_0, \delta)$ *on a system with action set* $\mathcal{A}$ transforms *the input sequence* $\sigma \in \mathcal{A}^\infty$ *into the output sequence* $\tau \in \mathcal{A}^\infty$*, notated as* $(q_0, \sigma) \Downarrow_S \tau$*, if and only if the following two constraints are met.*

1. $\forall q' \in Q : \forall \sigma' \in \mathcal{A}^\infty : \forall \tau' \in \mathcal{A}^\star : ((q_0, \sigma) \stackrel{\tau'}{\Longrightarrow}_S (q', \sigma')) \implies \tau' \preceq \tau$
2. $\forall \tau' \preceq \tau : \exists q' \in Q : \exists \sigma' \in \mathcal{A}^\infty : (q_0, \sigma) \stackrel{\tau'}{\Longrightarrow}_S (q', \sigma')$

When $(q_0, \sigma) \Downarrow_S \tau$, the first constraint ensures that automaton S on input σ outputs *only* prefixes of τ, while the second ensures that S outputs *every* prefix of τ.

2.4 Property Enforcement

We and several other authors have concurrently noted the importance of monitors obeying two abstract principles, which we call *soundness* and *transparency* [37, 24, 18]. A mechanism that purports to enforce a property $\hat{P}$ is *sound* when it ensures that observable outputs always obey $\hat{P}$; it is *transparent* when it preserves the semantics of executions that already obey $\hat{P}$. We call a sound and transparent mechanism an *effective* enforcer. Because effective enforcers are transparent, they may transform valid input sequences only into semantically equivalent output sequences, for some system-specific definition of semantic equivalence. When two executions $\sigma, \tau \in \mathcal{A}^\infty$ are semantically equivalent, we write $\sigma \cong \tau$. We place no restrictions on a relation of semantic equivalence except that it actually be an equivalence relation (i.e., reflexive, symmetric, and transitive), and that properties of interest $\hat{P}$ do not distinguish between semantically equivalent executions.

$$\forall \sigma, \tau \in \mathcal{A}^\infty : \sigma \cong \tau \implies (\hat{P}(\sigma) \iff \hat{P}(\tau)) \qquad \text{(Indistinguishability)}$$

When acting on a system with semantic equivalence relation $\cong$, we will call an effective enforcer an *effective$_{\cong}$ enforcer*. The formal definition of effective$_{\cong}$ enforcement is given below. Together, the first and second constraints in the following definition imply soundness; the first and third constraints imply transparency.

Definition 3 (Effective$_{\cong}$ Enforcement)
An automaton S with starting state q_0 effectively$_{\cong}$ enforces *a property $\hat{P}$ on a system with action set $\mathcal{A}$ and semantic equivalence relation $\cong$ if and only if* $\forall \sigma \in \mathcal{A}^{\infty} : \exists \tau \in \mathcal{A}^{\infty} :$

1. $(q_0, \sigma) \Downarrow_S \tau$,
2. $\hat{P}(\tau)$, *and*
3. $\hat{P}(\sigma) \implies \sigma \cong \tau$

In some situations, the system-specific equivalence relation $\cong$ complicates our theorems and proofs with little benefit. We have found that we can sometimes gain more insight into the enforcement powers of program monitors by limiting our analysis to systems in which the equivalence relation ($\cong$) is just syntactic equality ($=$). We call effective$_{\cong}$ enforcers operating on such systems *effective$_{=}$ enforcers*. To obtain a formal notion of effective$_{=}$ enforcement, we first need to define the "syntactic equality" of executions. Intuitively, $\sigma{=}\tau$ for any finite or infinite sequences σ and τ when every prefix of σ is a prefix of τ, and vice versa.

$$\forall \sigma, \tau \in \mathcal{A}^{\infty} : \sigma{=}\tau \iff (\forall \sigma' \preceq \sigma : \sigma' \preceq \tau \ \wedge \ \forall \tau' \preceq \tau : \tau' \preceq \sigma) \qquad \text{(Equality)}$$

An effective$_{=}$ enforcer is simply an effective$_{\cong}$ enforcer where the system-specific equivalence relation ($\cong$) is the system-unspecific equality relation ($=$).

Definition 4 (Effective$_=$ Enforcement)

An automaton S with starting state q_0 effectively$_=$ *enforces a property $\hat{P}$ on a system with action set $\mathcal{A}$ if and only if* $\forall\sigma \in \mathcal{A}^\infty : \exists\tau \in \mathcal{A}^\infty$:

1. $(q_0, \sigma) \Downarrow_S \tau$,
2. $\hat{P}(\tau)$, *and*
3. $\hat{P}(\sigma) \implies \sigma{=}\tau$

Because any two executions that are syntactically equal must be semantically equivalent, any property effectively$_=$ enforceable by some security automaton is also effectively$_\cong$ enforceable by that same automaton. Hence, an analysis of the set of properties effectively$_=$ enforceable by a particular kind of automaton is conservative; the set of properties effectively$_\cong$ enforceable by that same sort of automaton must be a superset of the effectively$_=$ enforceable properties.

Chapter 3

Policies Enforceable by Monitors

Now that we have set up a framework for formally reasoning about policies, properties, monitors (security automata), and enforcement, we can consider the space of properties enforceable by program monitors. In this chapter, we examine the enforcement powers of two types of monitors: a very simple but widely studied variety that we model with *truncation automata* (Section 3.1) and a more sophisticated variety that we model with *edit automata* (Section 3.2). We compare the properties enforceable by these two types of monitors and show that although the simple monitors can enforce only safety properties, it is possible to enforce some non-safety properties using more sophisticated monitors (Section 3.3).

3.1 Truncation Automata

We begin by demonstrating why it is often believed that program monitors enforce only safety properties: this belief is provably correct when considering a common but very limited type of monitor that we model by *truncation automata*. A truncation automaton has only two options when it intercepts an action from the target

program: it may accept the action and make it observable, or it may halt (i.e., truncate the action sequence of) the target program altogether. Schneider first defined this model of program monitors [48], and other authors have similarly focused on this simple, though limited, model when considering the properties enforceable by security automata [51, 32, 23]. Truncation-based monitors have been used successfully to enforce a rich set of interesting safety policies including access control [22], stack inspection [19, 3], software fault isolation [52, 20], Chinese Wall [14, 18, 23], and one-out-of-k authorization [23] policies.[1]

Although previous models of program monitors considered security automata to be invalid-sequence *recognizers* (a monitor simply halts the target when it recognizes a policy violation), we model program monitors more generally as sequence *transformers*. This shift enables us to define more sophisticated monitors such as edit automata (Section 3.2) but also makes it important for us to recast the previous work on truncation automata to fit our model. Moving the analysis into our formal model allows us to refine previous work by uncovering the single computable safety property unenforceable by any truncation (or edit) automaton. Considering truncation automata directly in our model also enables us to precisely compare the enforcement powers of truncation and edit automata.

3.1.1 Definition

A truncation automaton T is a finite or countably infinite state machine (Q, q_0, δ) that is defined with respect to some system with action set $\mathcal{A}$. As usual, Q specifies the possible automaton states, and q_0 is the initial state. The complete function

[1] Although some of the cited work considers monitors with powers beyond truncation, it also specifically studies many policies that can be enforced by monitors that only have the power to truncate.

$\delta : Q \times \mathcal{A} \to Q \cup \{halt\}$ specifies the transition function for the automaton and indicates either that the automaton should accept the current input action and move to a new state (when the return value is a new state), or that the automaton should halt the target program (when the return value is *halt*). For the sake of determinacy, we require that $halt \notin Q$. The operational semantics of truncation automata are formally specified by the following rules.

$\boxed{(q, \sigma) \xrightarrow{\tau}_T (q', \sigma')}$

$$(q, \sigma) \xrightarrow{a}_T (q', \sigma') \qquad \text{(T-Step)}$$

if $\sigma = a; \sigma'$

and $\delta(q, a) = q'$

$$(q, \sigma) \xrightarrow{\cdot}_T (q, \cdot) \qquad \text{(T-Stop)}$$

if $\sigma = a; \sigma'$

and $\delta(q, a) = halt$

As described in Section 2.3, we extend the single-step relation to a multi-step relation using standard reflexivity and transitivity rules.

3.1.2 Enforceable Properties

Let us consider a lower bound on the effective$_{\cong}$ enforcement powers of truncation automata. Any property that is effectively$_{=}$ enforceable by a truncation automaton is also effectively$_{\cong}$ enforceable by that same automaton, so we can develop a lower bound on properties effectively$_{\cong}$ enforceable by examining which properties are effectively$_{=}$ enforceable.

When given as input some $\sigma \in \mathcal{A}^\infty$ such that $\hat{P}(\sigma)$, a truncation automaton that effectively$_{=}$ enforces $\hat{P}$ must output σ. However, the automaton must also truncate

every invalid input sequence into a valid output. Any truncation of an invalid input prevents the automaton from accepting all the actions in a valid extension of that input. Therefore, truncation automata cannot effectively$_=$ enforce any property in which an invalid execution can be a prefix of a valid execution. This is exactly the definition of safety properties, so it is clear that truncation automata effectively$_=$ enforce only safety properties.

Past research claimed to equate the enforcement power of truncation automata with the set of computable safety properties [51, 32]. We improve previous work by showing that there is exactly one computable safety property unenforceable by any sound security automaton: the unsatisfiable safety property that considers all executions invalid. A monitor in our framework cannot enforce such a property because there is no valid output sequence that could be produced in response to an invalid input sequence. To prevent this case and to ensure that truncation automata can behave correctly on targets that generate no actions, we require that the empty sequence satisfies any property we are interested in enforcing. We often use the term *reasonable* to describe computable properties $\hat{P}$ such that $\hat{P}(\cdot)$.

Definition 5 (Reasonable Property)

A property $\hat{P}$ on a system with action set $\mathcal{A}$ is reasonable *if and only the following conditions hold.*

1. $\hat{P}(\cdot)$
2. $\forall \sigma \in \mathcal{A}^\star : \hat{P}(\sigma)$ *is decidable*

The following theorem states that truncation automata effectively$_=$ enforce exactly the set of reasonable safety properties.

Theorem 6 (Effective$_=$ T^∞-Enforcement)

A property $\hat{P}$ on a system with action set $\mathcal{A}$ can be effectively$_=$ enforced by some truncation automaton T if and only if the following constraints are met.

1. $\forall \sigma \in \mathcal{A}^\infty : \neg\hat{P}(\sigma) \implies \exists \sigma' \preceq \sigma : \forall \tau \succeq \sigma' : \neg\hat{P}(\tau)$ (SAFETY)
2. $\hat{P}(\cdot)$
3. $\forall \sigma \in \mathcal{A}^\star : \hat{P}(\sigma)$ *is decidable*

Proof **(If Direction)** We construct a truncation automaton T that effectively$_=$ enforces any such $\hat{P}$ as follows.

- States: $Q = \mathcal{A}^\star$ (the sequence of actions seen so far)
- Start state: $q_0 = \cdot$ (the empty sequence)
- Transition function: $\delta(\sigma, a) = \begin{cases} \sigma; a & \text{if } \hat{P}(\sigma; a) \\ \mathit{halt} & \text{otherwise} \end{cases}$

  This transition function is computable because $\hat{P}$ is decidable over all finite-length executions.

T maintains the invariant $I_{\hat{P}}(q)$ on states $q = \sigma$ that exactly σ has been output from T, $(q_0, \sigma) \Downarrow_T \sigma$, and $\forall \sigma' \preceq \sigma : \hat{P}(\sigma')$. The automaton can initially establish $I_{\hat{P}}(q_0)$ because $q_0 = \cdot$, $(q_0, \cdot) \Downarrow_T \cdot$, and $\hat{P}(\cdot)$. A simple inductive argument on the length of σ suffices to show that the invariant is maintained for all (finite-length) prefixes of all inputs.

Let $\sigma \in \mathcal{A}^\infty$ be the input to T. If $\neg\hat{P}(\sigma)$ then by the safety condition in the theorem statement, $\exists \sigma' \preceq \sigma . \neg\hat{P}(\sigma')$. By $I_{\hat{P}}(\sigma')$, T can never enter the state for this σ' and must therefore halt on input σ. Let τ be the final state reached on input σ.

By $I_{\hat{P}}(\tau)$ and the fact that T halts (ceases to make transitions) after reaching state τ, we have $\hat{P}(\tau)$ and $(q_0, \sigma) \Downarrow_T \tau$.

If, on the other hand, $\hat{P}(\sigma)$ then suppose for the sake of obtaining a contradiction that T on input σ does not accept and output every action of σ. By the definition of its transition function, T must halt in some state σ' when examining some action a (where $\sigma'; a \preceq \sigma$) because $\neg\hat{P}(\sigma'; a)$. Combining this with the safety condition given in the theorem statement implies that $\neg\hat{P}(\sigma)$, which is a contradiction. Hence, T accepts and outputs every action of σ when $\hat{P}(\sigma)$, so $(q_0, \sigma) \Downarrow_T \sigma$. In all cases, T effectively$_=$ enforces $\hat{P}$.

(Only-If Direction) Consider any $\sigma \in \mathcal{A}^\infty$ such that $\neg\hat{P}(\sigma)$ and suppose for the sake of obtaining a contradiction that $\forall\sigma' \preceq \sigma : \exists\tau \succeq \sigma' : \hat{P}(\tau)$. Then for all prefixes σ' of σ, T must accept and output every action of σ' because σ' may be extended to the valid input τ, which must be emitted verbatim. This implies by the definition of $\Downarrow_T$ that $(q_0, \sigma) \Downarrow_T \sigma$ (where q_0 is the initial state of T), which is a contradiction because T cannot effectively$_=$ enforce $\hat{P}$ on σ when $\neg\hat{P}(\sigma)$ and $(q_0, \sigma) \Downarrow_T \sigma$. Hence, our assumption was incorrect and the first constraint given in the theorem must hold.

Also, if $\neg\hat{P}(\cdot)$ then T cannot effectively$_=$ enforce $\hat{P}$ on an empty execution because $(q_0, \cdot) \Downarrow_T \cdot$ for all T. Therefore, $\hat{P}(\cdot)$.

Finally, given $\sigma \in \mathcal{A}^\star$, we can decide $\hat{P}(\sigma)$ by checking whether T outputs exactly σ on input σ. Because T effectively$_=$ enforces $\hat{P}$, $\hat{P}(\sigma) \iff (q_0, \sigma) \Downarrow_T \sigma$. This is a decidable procedure because T's transition function is computable and σ has finite length. ■

We next delineate the properties effectively$_{\cong}$ enforceable by truncation automata. As mentioned above, the set of properties truncation automata effectively$_{=}$ enforce provides a lower bound for the set of effectively$_{\cong}$ enforceable properties; a candidate upper bound is the set of properties $\hat{P}$ that satisfy the following extended safety constraint.

$$\forall \sigma \in \mathcal{A}^{\infty} : \neg \hat{P}(\sigma) \implies \exists \sigma' \preceq \sigma : \forall \tau \succeq \sigma' : (\neg \hat{P}(\tau) \vee \tau \cong \sigma') \qquad \text{(T-Safety)}$$

This is an upper bound because a truncation automaton T that effectively$_{\cong}$ enforces $\hat{P}$ must halt at some finite point (having output σ') when its input (σ) violates $\hat{P}$; otherwise, T accepts every action of the invalid input. When T halts, all extensions (τ) of its output must either violate $\hat{P}$ or be equivalent to its output; otherwise, there is a valid input for which T fails to output an equivalent sequence.

Actually, as the following theorem shows, this upper bound is almost tight. We simply have to add computability restrictions on the property to ensure that a truncation automaton can decide when to halt the target.

Theorem 7 (Effective$_{\cong}$ T^{∞}-Enforcement)
A property $\hat{P}$ on a system with action set $\mathcal{A}$ can be effectively$_{\cong}$ enforced by some truncation automaton T if and only if there exists a decidable predicate D over $\mathcal{A}^{\star}$ such that the following constraints are met.

1. $\forall \sigma \in \mathcal{A}^{\infty} : \neg \hat{P}(\sigma) \implies \exists \sigma' \preceq \sigma : D(\sigma')$
2. $\forall (\sigma'; a) \in \mathcal{A}^{\star} : D(\sigma'; a) \implies (\hat{P}(\sigma') \wedge \forall \tau \succeq (\sigma'; a) : \hat{P}(\tau) \implies \tau \cong \sigma')$
3. $\neg D(\cdot)$

Proof **(If Direction)** We first note that the first and third constraints imply that $\hat{P}(\cdot)$, as there can be no prefix σ' of the empty sequence such that $D(\sigma')$. We next construct a truncation automaton T that, given decidable predicate D and property $\hat{P}$, effectively$_{\cong}$ enforces $\hat{P}$ when the constraints in the theorem statement are met.

- States: $Q = \mathcal{A}^\star$ (the sequence of actions seen so far)
- Start state: $q_0 = \cdot$ (the empty sequence)
- Transition function: $\delta(\sigma, a) = \begin{cases} \sigma; a & \text{if } \neg D(\sigma; a) \\ \mathit{halt} & \text{otherwise} \end{cases}$

  This transition function is computable because D is decidable.

T maintains the invariant $I_{\hat{P}}(q)$ on states $q = \sigma$ that exactly σ has been output from T, $(q_0, \sigma) \Downarrow_T \sigma$, and $\forall \sigma' \preceq \sigma : \neg D(\sigma')$. The automaton can initially establish $I_{\hat{P}}(q_0)$ because $q_0 = \cdot$, $(q_0, \cdot) \Downarrow_T \cdot$, and $\neg D(\cdot)$. A simple inductive argument on the length of σ suffices to show that the invariant is maintained for all (finite-length) prefixes of all inputs.

Let $\sigma \in \mathcal{A}^\infty$ be the input to T. We first consider the case where $\neg\hat{P}(\sigma)$ and show that T effectively$_{\cong}$ enforces $\hat{P}$ on σ. By constraint 1 in the theorem statement, $\exists \sigma' \preceq \sigma : D(\sigma')$, so $I_{\hat{P}}$ ensures that T must halt when σ is input (before entering state σ'). Let τ be the final state T reaches on input σ before halting when considering action a. By $I_{\hat{P}}(\tau)$, we have $(q_0, \sigma) \Downarrow_T \tau$. Also, since $D(\tau; a)$ forced T to halt, constraint 2 in the theorem statement ensures that $\hat{P}(\tau)$.

We split the case where $\hat{P}(\sigma)$ into two subcases. If T never truncates input σ then T outputs every prefix of σ and only prefixes of σ, so by the definition of $\Downarrow_T$, $(q_0, \sigma) \Downarrow_T \sigma$. Because $\hat{P}(\sigma)$ and $\sigma \cong \sigma$, T effectively$_{\cong}$ enforces $\hat{P}$ in this subcase. On the other hand, if T truncates input σ, it does so in some state

σ' while making a transition on action a (hence, $\sigma'; a \preceq \sigma$) because $D(\sigma'; a)$. In this subcase, $I_{\hat{P}}(\sigma')$ implies $(q_0, \sigma) \Downarrow_T \sigma'$. Also, since $D(\sigma'; a)$ forced T to halt, constraint 2 in the theorem statement ensures that $\hat{P}(\sigma')$ and $\sigma' \cong \sigma$. Therefore, T correctly effectively$_{\cong}$ enforces $\hat{P}$ in all cases.

(Only-If Direction) Given some truncation automaton T, we define D over $\mathcal{A}^\star$. Let $D(\cdot)$ be false, and for all $(\sigma; a) \in \mathcal{A}^\star$ let $D(\sigma; a)$ be true if and only if T outputs exactly σ on input $\sigma; a$ (when run to completion). Because the transition function of T is computable and D is only defined over finite sequences, D is a decidable predicate. Moreover, because T effectively$_{\cong}$ enforces $\hat{P}$, if it outputs exactly σ on input $\sigma; a$ then the fact that T halts rather than accepting a, combined with the definition of effective$_{\cong}$ enforcement, implies that $\hat{P}(\sigma) \wedge \forall \tau \succeq \sigma; a : \hat{P}(\tau) \implies \tau \cong \sigma$. Our definition of D thus satisfies the second constraint enumerated in the theorem.

Finally, consider any $\sigma \in \mathcal{A}^\infty$ such that $\neg\hat{P}(\sigma)$ and suppose for the sake of obtaining a contradiction that $\forall \sigma' \preceq \sigma : \neg D(\sigma')$. Then by our definition of D, T cannot halt on any prefix of σ, so it must accept every action in every prefix. This implies by the definition of $\Downarrow_T$ that $(q_0, \sigma) \Downarrow_T \sigma$ (where q_0 is the initial state of T), which is a contradiction because T cannot effectively$_{\cong}$ enforce $\hat{P}$ on σ when $\neg\hat{P}(\sigma)$ and $(q_0, \sigma) \Downarrow_T \sigma$. Hence, our assumption was incorrect and the first constraint given in the theorem must also hold. ■

On practical systems, it is likely uncommon that the property requiring enforcement and the system's relation of semantic equivalence are so broadly defined that some invalid execution has a prefix that not only can be extended to a valid execution, but that is also equivalent to *all* valid extensions of the prefix. We therefore

consider the set of properties detailed in the theorem of Effective$_=$ T^∞-Enforcement (i.e., reasonable safety properties) more indicative of the true enforcement power of truncation automata.

3.2 Edit Automata

We now consider the enforcement capabilities of a stronger sort of security automaton called the *edit automaton*. We analyze the enforcement powers of edit automata and find that they can effectively$_=$ enforce an interesting, new class of properties that we call *infinite renewal* properties.

3.2.1 Definition

An *edit automaton* E is a triple (Q, q_0, δ) defined with respect to some system with action set $\mathcal{A}$. As with truncation automata, Q is the possibly countably infinite set of states, and q_0 is the initial state. In contrast to truncation automata, the complete transition function δ of an edit automaton has the form $\delta : Q \times \mathcal{A} \to Q \times (\mathcal{A} \cup \{\cdot\})$. The transition function specifies, when given a current state and input action, a new state to enter and either an action to *insert* into the output stream (without consuming the input action) or the empty sequence to indicate that the input action should be *suppressed* (i.e., consumed from the input without being made observable). In other work, we have defined edit automata that can additionally perform the following transformations in a single step: insert a finite sequence of actions, accept the current input action, or halt the target [38]. However, all of these transformations can be expressed in terms of suppressing and inserting single actions. For example, an edit automaton can halt a target by suppressing all future actions

of the target; an edit automaton accepts an action by inserting and then suppressing that action (first making the action observable and then consuming it from the input). Although in practice these transformations would each be performed in a single step, we have found the minimal operational semantics containing only the two rules shown below more amenable to formal reasoning. Explicitly including the additional rules in the model would not invalidate any of our results.

$\boxed{(q, \sigma) \xrightarrow{\tau}_E (q', \sigma')}$

$$(q, \sigma) \xrightarrow{a'}_E (q', \sigma) \qquad \text{(E-Ins)}$$

if $\sigma = a; \sigma'$

and $\delta(q, a) = (q', a')$

$$(q, \sigma) \xrightarrow{\cdot}_E (q', \sigma') \qquad \text{(E-Sup)}$$

if $\sigma = a; \sigma'$

and $\delta(q, a) = (q', \cdot)$

As with truncation automata, we extend the single-step semantics of edit automata to a multi-step semantics with the rules for reflexivity and transitivity.

3.2.2 Enforceable Properties

Edit automata are powerful property enforcers because they can suppress a sequence of potentially illegal actions and later, if the sequence is determined to be legal, just re-insert it. Essentially, the monitor feigns to the target that its requests are being accepted, although none actually are, until the monitor can confirm that the sequence of feigned actions is valid. At that point, the monitor inserts all of the actions it previously feigned accepting. This is the same idea implemented by intentions files in database transactions [44]. Monitoring systems like virtual

machines can also be used in this way, feigning execution of a sequence of the target's actions and only making the sequence observable when it is known to be valid.

As we did for truncation automata, we develop a lower bound on the set of properties that edit automata effectively$_{\cong}$ enforce by considering the properties they effectively$_{=}$ enforce. Using the above-described technique of suppressing invalid inputs until the monitor determines that the suppressed input obeys a property, edit automata can effectively$_{=}$ enforce any reasonable *infinite renewal* (or simply *renewal*) property. A renewal property is one in which every valid infinite-length sequence has infinitely many valid prefixes, and conversely, every invalid infinite-length sequence has only finitely many valid prefixes. For example, a property $\hat{P}$ may be satisfied only by executions that contain the action a. This is a renewal property because valid infinite-length executions contain an infinite number of valid prefixes (in which a appears) while invalid infinite-length executions contain only a finite number of valid prefixes (in fact, zero). This $\hat{P}$ is also a liveness property because any invalid finite execution can be made valid simply by appending the action a. Although edit automata cannot enforce this $\hat{P}$ because $\neg\hat{P}(\cdot)$, in Section 3.3.2 we will recast this example as a reasonable "eventually audits" policy and show several more detailed examples of renewal properties enforceable by edit automata.

We formally deem a property $\hat{P}$ an infinite renewal property on a system with action set $\mathcal{A}$ if and only if the following is true.

$$\forall \sigma \in \mathcal{A}^{\omega} : \hat{P}(\sigma) \iff \{\sigma' \preceq \sigma \mid \hat{P}(\sigma')\} \textit{ is an infinite set} \qquad (\textsc{Renewal}_1)$$

It will often be easier to reason about renewal properties without relying on infinite set cardinality. We make use of the following equivalent definition in formal analyses.

$$\forall \sigma \in \mathcal{A}^\omega : \hat{P}(\sigma) \iff (\forall \sigma' \preceq \sigma : \exists \tau \preceq \sigma : \sigma' \preceq \tau \wedge \hat{P}(\tau)) \qquad (\text{RENEWAL}_2)$$

If we are given a reasonable renewal property $\hat{P}$, we can construct an edit automaton that effectively$_=$ enforces $\hat{P}$ using the technique of feigning acceptance (i.e., suppressing actions) until the automaton has seen some legal prefix of the input (at which point the suppressed actions can be made observable). This technique ensures that the automaton eventually outputs every valid prefix, and only valid prefixes, of any input execution. Because $\hat{P}$ is a renewal property, the automaton therefore outputs all prefixes, and only prefixes, of a valid input while outputting only the longest valid prefix of an invalid input. Hence, the automaton correctly effectively$_=$ enforces $\hat{P}$. The following theorem formally states this result.

Theorem 8 (Lower Bound Effective$_=$ E^∞-Enforcement)
A property $\hat{P}$ on a system with action set $\mathcal{A}$ can be effectively$_=$ enforced by some edit automaton E if the following constraints are met.

1. $\forall \sigma \in \mathcal{A}^\omega : \hat{P}(\sigma) \iff (\forall \sigma' \preceq \sigma : \exists \tau \preceq \sigma : \sigma' \preceq \tau \wedge \hat{P}(\tau))$ (RENEWAL_2)
2. $\hat{P}(\cdot)$
3. $\forall \sigma \in \mathcal{A}^\star : \hat{P}(\sigma)$ *is decidable*

Proof We construct an edit automaton E that effectively$_=$ enforces any such $\hat{P}$ as follows.

- States: $Q = \mathcal{A}^\star \times \mathcal{A}^\star \times \{0,1\}$ (the sequence of actions output so far, the sequence of actions currently suppressed, and a flag indicating whether the suppressed actions need to be inserted)
- Start state: $q_0 = (\cdot, \cdot, 0)$ (nothing has been output or suppressed)
- Transition function:

$$\delta((\tau,\sigma,f),a) = \begin{cases} ((\tau,\sigma;a,0),\cdot) & \text{if } f = 0 \wedge \neg\hat{P}(\tau;\sigma;a) \\ ((\tau;a',\sigma',1),a') & \text{if } f = 0 \wedge \hat{P}(\tau;\sigma;a) \wedge \sigma; a{=}a';\sigma' \\ ((\tau;a',\sigma',1),a') & \text{if } f = 1 \wedge \sigma{=}a';\sigma' \\ ((\tau,\cdot,0),\cdot) & \text{if } f = 1 \wedge \sigma{=}\cdot \end{cases}$$

  This transition function is computable because $\hat{P}$ is decidable over all finite-length executions.

E maintains the invariant $I_{\hat{P}}(q)$ on states $q = (\tau, \sigma, 0)$ that exactly τ has been output, $\tau;\sigma$ is the input that has been processed, $(q_0, \tau;\sigma) \Downarrow_E \tau$, and τ is the longest prefix of $\tau;\sigma$ such that $\hat{P}(\tau)$. Similarly, E maintains $I_{\hat{P}}(q)$ on states $q = (\tau, \sigma, 1)$ that exactly τ has been output, all of $\tau;\sigma$ except the action on which E is currently making a transition is the input that has been processed, $\hat{P}(\tau;\sigma)$, and E will finish processing the current action when all of $\tau;\sigma$ has been output, the current action has been suppressed, and E is in state $(\tau;\sigma,\cdot,0)$. The automaton can initially establish $I_{\hat{P}}(q_0)$ because $q_0 = (\cdot,\cdot,0)$, $(q_0,\cdot) \Downarrow_E \cdot$, and $\hat{P}(\cdot)$. A simple inductive argument on the transition relation suffices to show that E maintains the invariant in every state it reaches.

Let $\sigma \in \mathcal{A}^\infty$ be the input to the automaton E. If $\neg\hat{P}(\sigma)$ and $\sigma \in \mathcal{A}^\star$ then by the automaton invariant, E consumes all of input σ and halts in some state $(\tau, \sigma', 0)$ such that $(q_0, \sigma) \Downarrow_E \tau$ and $\hat{P}(\tau)$. Hence, E effectively$_=$ enforces $\hat{P}$ in this case. If $\neg\hat{P}(\sigma)$ and $\sigma \in \mathcal{A}^\omega$ then by the renewal condition in the theorem statement, there

must be some prefix σ' of σ such that for all longer prefixes τ of σ, $\neg\hat{P}(\tau)$. Thus, by the transition function of E, the invariant of E, and the definition of $\Downarrow_E$, E on input σ outputs only some finite τ' such that $\hat{P}(\tau')$ and $(q_0, \sigma) \Downarrow_E \tau'$ (and E suppresses all actions in σ after outputting τ').

Next consider the case where $\hat{P}(\sigma)$. If $\sigma \in \mathcal{A}^\star$ then by the automaton invariant, E on input σ must halt in state $(\sigma, \cdot, 0)$, where $(q_0, \sigma) \Downarrow_E \sigma$. E thus effectively$_=$ enforces $\hat{P}$ in this case. If $\hat{P}(\sigma)$ and $\sigma \in \mathcal{A}^\omega$ then the renewal constraint and the automaton invariant ensure that E on input σ outputs every prefix of σ and only prefixes of σ. Hence, $(q_0, \sigma) \Downarrow_E \sigma$. In all cases, E correctly effectively$_=$ enforces $\hat{P}$.

■

It would be reasonable to expect that the set of renewal properties also represents an upper bound on the properties effectively$_=$ enforceable by edit automata. After all, an effective$_=$ automaton cannot output an infinite number of valid prefixes of an invalid infinite-length input σ without outputting σ itself. In addition, on a valid infinite-length input τ, an effective$_=$ automaton must output infinitely many prefixes of τ, and whenever it finishes processing an input action, its output must be a *valid* prefix of τ because there may be no more input (i.e., the target may not generate more actions).

However, there is a corner case in which an edit automaton can effectively$_=$ enforce a valid infinite-length execution τ that has only finitely many valid prefixes. If, after processing a prefix of τ, the automaton can decide that τ is the only valid extension of this prefix, then the automaton can cease processing input and enter an infinite loop to insert the remaining actions of τ. While in this infinite loop, the automaton need not output infinitely many valid prefixes, since it is certain to

be able to extend the current (invalid) output into an infinite-length valid output sequence.

The following theorem presents the tight boundary for effective$_=$ enforcement of properties by edit automata, including the corner case described above. Because we believe that the corner case adds relatively little to the enforcement capabilities of edit automata, we only sketch the proof.

Theorem 9 (Effective$_=$ E^∞-Enforcement)
A property $\hat{P}$ on a system with action set $\mathcal{A}$ can be effectively$_=$ enforced by some edit automaton E if and only if the following constraints are met.

1. $\forall \sigma \in \mathcal{A}^\omega : \hat{P}(\sigma) \iff$
$$\left(\begin{array}{l} \forall \sigma' \preceq \sigma : \exists \tau \preceq \sigma : \sigma' \preceq \tau \wedge \hat{P}(\tau) \\ \vee\ \hat{P}(\sigma) \wedge \\ \quad \exists \sigma' \preceq \sigma : \forall \tau \succeq \sigma' : \hat{P}(\tau) \implies \tau = \sigma \wedge \\ \quad \textit{the existence and actions of } \sigma \\ \quad \textit{are computable from } \sigma' \end{array} \right)$$
2. $\hat{P}(\cdot)$
3. $\forall \sigma \in \mathcal{A}^\star : \hat{P}(\sigma)$ *is decidable*

Proof (sketch): (If Direction) We sketch the construction of an edit automaton E that effectively$_=$ enforces any such $\hat{P}$ as follows.

- States: $Q = \mathcal{A}^\star \times \mathcal{A}^\star$ (the sequence of actions output so far paired with the sequence of actions suppressed since the previous insertion)
- Start state: $q_0 = (\cdot, \cdot)$ (nothing has been output or suppressed)
- Transition function (for simplicity, we abbreviate δ):
  Consider processing the action a in state (τ', σ').

(A) If we can compute from $\tau'; \sigma'$ the existence and actions of some $\sigma \in \mathcal{A}^\omega$ such that $\forall \tau \succeq (\tau'; \sigma') : \hat{P}(\tau) \implies \tau = \sigma$, enter an infinite loop that inserts one by one all actions necessary to output every prefix of σ.

(B) Otherwise, if $\hat{P}(\tau'; \sigma'; a)$ then insert $\sigma'; a$ (one action at a time), suppress a, and continue in state $(\tau'; \sigma'; a, \cdot)$.

(C) Otherwise, suppress a and continue in state $(\tau', \sigma'; a)$.

This automaton is an informal version of the one constructed in the "if" direction of the proof of Theorem 8, except for the addition of transition (A), and E effectively$_=$ enforces $\hat{P}$ for the same reasons given there. The only difference is that E can insert an infinite sequence of actions if it computes that only that sequence of actions can extend the current input to satisfy $\hat{P}$. In this case, E continues to effectively$_=$ enforce $\hat{P}$ because its output satisfies $\hat{P}$ and equals any valid input sequence.

(Only-If Direction) Consider any $\sigma \in \mathcal{A}^\omega$ such that $\hat{P}(\sigma)$. By the definition of effective$_=$ enforcement, $(q_0, \sigma) \Downarrow_E \sigma$, where q_0 is the initial state of E. By the definitions of $\Downarrow_E$ and $=$, E must output all prefixes of σ and only prefixes of σ when σ is input. Assume for the sake of obtaining a contradiction that the extended renewal constraint is untrue for σ. This implies that there is some valid prefix σ' of σ after which all longer prefixes of σ violate $\hat{P}$. After outputting σ' on input σ', E cannot output any prefix of σ without outputting every prefix of σ (if it did, its output would violate $\hat{P}$). But because the extended renewal constraint does not hold on σ by assumption, either (1) more than one valid execution will always extend the automaton's input or (2) E can never compute or emit all prefixes of σ. Therefore, E cannot output every prefix of σ after outputting σ', so E fails to

effectively$_=$ enforce $\hat{P}$ on this σ. Our assumption was therefore incorrect, and the renewal constraint must hold.

Next consider any $\sigma \in \mathcal{A}^\omega$ such that $\neg\hat{P}(\sigma)$. The extended portion of the renewal constraint trivially holds because $\neg\hat{P}(\sigma)$. Assume for the sake of obtaining a contradiction that the rest of the renewal constraint does not hold on σ, implying that there are an infinite number of prefixes of σ that satisfy $\hat{P}$. Because E is an effective$_=$ enforcer and can only enforce $\hat{P}$ on sequences obeying $\hat{P}$ by emitting them verbatim, E must eventually output every prefix of σ and only prefixes of σ when σ is input. Hence, $(q_0, \sigma) \Downarrow_E \sigma$, which is a contradiction because E effectively$_=$ enforces $\hat{P}$ and $\neg\hat{P}(\sigma)$. Our assumption that the renewal constraint does not hold is therefore incorrect.

Also, $\hat{P}(\cdot)$ because E could otherwise not effectively$_=$ enforce $\hat{P}$ when input the empty sequence.

Finally, we decide $\hat{P}(\sigma)$ for all $\sigma \in \mathcal{A}^\star$ using the same procedure described in the "Only-If" direction of the proof of Theorem 6. ■

We have found it difficult to precisely characterize the properties that are effectively$_\cong$ enforceable by edit automata. Unfortunately, the simplest way to specify this set appears to be to encode the semantics of edit automata into recursive functions that operate over streams of actions. Then, we can reason about the relationship between input and output sequences of such functions just as the definition of effective$_\cong$ enforcement requires us to reason about the relationship between input and output sequences of automata. Our final theorem takes this approach; we present it for completeness.

Theorem 10 (Effective$_{\cong}$ E^{∞}-Enforcement)

Let repls *be a decidable function* $\text{repls} : \mathcal{A}^{\star} \times \mathcal{A}^{\star} \rightarrow \mathcal{A} \cup \{\cdot\}$. *Then* $R^{\star}_{\text{repls}}$ *is a decidable function* $R^{\star}_{\text{repls}} : \mathcal{A}^{\star} \times \mathcal{A}^{\star} \times \mathcal{A}^{\star} \rightarrow \mathcal{A}^{\star}$ *parameterized by* repls *and inductively defined as follows, where all metavariables are universally quantified.*

- $R^{\star}_{\text{repls}}(\cdot, \sigma, \tau) = \tau$
- $(\text{repls}(\sigma; a, \tau) = \cdot) \implies R^{\star}_{\text{repls}}(a; \sigma', \sigma, \tau') = R^{\star}_{\text{repls}}(\sigma', \sigma; a, \tau')$
- $(\text{repls}(\sigma; a, \tau) = a') \implies R^{\star}_{\text{repls}}(a; \sigma', \sigma, \tau') = R^{\star}_{\text{repls}}(a; \sigma', \sigma, \tau'; a')$

A property $\hat{P}$ *on a system with action set* $\mathcal{A}$ *can be effectively*$_{\cong}$ *enforced by some edit automaton* E *if and only if there exists a decidable* repls *function (as described above) such that for all (input sequences)* $\sigma \in \mathcal{A}^{\infty}$ *there exists (output sequence)* $\tau \in \mathcal{A}^{\infty}$ *such that the following constraints are met.*

1. $\forall \sigma' \preceq \sigma : \forall \tau' \in \mathcal{A}^{\star} : (R^{\star}_{\text{repls}}(\sigma', \cdot, \cdot) = \tau') \implies \tau' \preceq \tau$
2. $\forall \tau' \preceq \tau : \exists \sigma' \preceq \sigma : R^{\star}_{\text{repls}}(\sigma', \cdot, \cdot) = \tau'$
3. $\hat{P}(\tau)$
4. $\hat{P}(\sigma) \implies \sigma \cong \tau$

Proof (sketch): Intuitively, $\text{repls}(\sigma, \tau) = a$ (or $\cdot$) iff a is the next action to be output (or suppressed) by an edit automaton when σ is the automaton input and τ is the automaton output so far. Also, $R^{\star}_{\text{repls}}(\sigma, \sigma', \tau') = \tau$ iff the overall output of an edit automaton whose transition function is guided by repls is τ when σ remains to be processed, σ' has already been processed, and τ' has already been output.

(If Direction) Given repls, we construct an edit automaton E that effectively$_{\cong}$ enforces any such $\hat{P}$ as follows.

- States: $Q = \mathcal{A}^\star \times \mathcal{A}^\star$ (the input processed and the output emitted so far)
- Start state: $q_0 = (\cdot, \cdot)$ (nothing processed or output)
- Transition function: $\delta((\sigma, \tau), a) = \begin{cases} ((\sigma, \tau; a'), a') & \text{if } \text{repls}(\sigma; a, \tau) = a' \\ ((\sigma; a, \tau), \cdot) & \text{otherwise} \end{cases}$

For all prefixes σ' of the input σ to E, E emits a τ' such that $R^\star_{\text{repls}}(\sigma', \cdot, \cdot) = \tau'$. The proof is by induction on the length of σ', using the definition of $R^\star_{\text{repls}}$. Then, by the constraints in the theorem statement and the definitions of $\Downarrow_E$ and effective$_{\cong}$ enforcement, E effectively$_{\cong}$ enforces $\hat{P}$.

(Only-If Direction) Define $\text{repls}(\sigma, \tau)$ as follows. Run E on input σ until τ is output (if τ is not a prefix of the output then arbitrarily define $\text{repls}(\sigma, \tau) = \cdot$), and then continue running E until either all input is consumed (i.e., suppressed) or another action a' is output. In the former case, let $\text{repls}(\sigma, \tau) = \cdot$ and in the latter case $\text{repls}(\sigma, \tau) = a'$. D is decidable because σ and τ have finite lengths and the transition function of E is computable.

By the definitions of repls and $R^\star_{\text{repls}}$, we have the following. $\forall \sigma, \tau \in \mathcal{A}^\star$: $(R^\star_{\text{repls}}(\sigma, \cdot, \cdot) = \tau) \iff (\exists q' : (q_0, \sigma) \stackrel{\tau}{\Longrightarrow}_E (q', \cdot))$, where q_0 is the initial state of E. Combining this with the definition of $\Downarrow_E$ and the fact that E effectively$_{\cong}$ enforces $\hat{P}$ ensures that all of the constraints given in the theorem statement are satisfied.

■

As with truncation automata, we believe that the theorems related to edit automata acting as effective$_{=}$ enforcers more naturally capture their inherent power

than does the theorem of effective$_{\cong}$ enforcement. Effective$_{=}$ enforcement provides an elegant lower bound for what can be effectively$_{\cong}$ enforced in practice.

Limitations In addition to standard assumptions of program monitors, such as that a target cannot circumvent or corrupt a monitor, our theoretical model makes assumptions particularly relevant to edit automata that are sometimes violated in practice. Most importantly, our model assumes that security automata have the same computational capabilities as the system that observes the monitor's output. If an action violates this assumption by requiring an outside system in order to be executed, it cannot be "feigned" (i.e., suppressed) by the monitor. For example, it would be impossible for a monitor to feign sending email, wait for the target to receive a response to the email, test whether the target does something invalid with the response, and then decide to "undo" sending email in the first place. Here, the action for sending email has to be made observable to systems outside of the monitor's control in order to be executed, so this is an unsuppressible action. A similar limitation arises with time-dependent actions, where an action cannot be feigned (i.e., suppressed) because it may behave differently if made observable later. In addition to these sorts of unsuppressible actions, a system may contain actions uninsertable by monitors because, for example, the monitors lack access to secret keys that must be passed as parameters to the actions. In the future, we plan to explore the usefulness of including sets of unsuppressible and uninsertable actions in the specification of systems. We might be able to harness some of our other work [38], which defined security automata limited to inserting (insertion automata) or suppressing (suppression automata) actions, toward this goal.

3.3 Infinite Renewal Properties

In this section, we examine some interesting aspects of the class of infinite renewal properties. We compare renewal properties to safety and liveness properties and provide several examples of useful renewal properties that are neither safety nor liveness properties.

3.3.1 Renewal, Safety, and Liveness

The most obvious way in which safety and infinite renewal properties differ is that safety properties place restrictions on finite executions (invalid finite executions must have some prefix after which all extensions are invalid), while renewal properties place no restrictions on finite executions. Thus, if we consider systems that only exhibit finite executions, edit automata can enforce *every* reasonable property [38]. Without infinite-length executions, every property is a renewal property.

Moreover, an infinite-length renewal execution can be valid even if it has infinitely many invalid prefixes (as long as it also has infinitely many valid prefixes), but a valid safety execution can contain no invalid prefixes. Similarly, although invalid infinite-length renewal executions can have prefixes that alternate a finite number of times between being valid and invalid, invalid safety executions must contain some finite prefix before which all prefixes are valid and after which all prefixes are invalid. Hence, every safety property is a renewal property. Given any system with action set $\mathcal{A}$, it is easy to construct a non-safety renewal property $\hat{P}$ by choosing an element a in $\mathcal{A}$ and letting $\hat{P}(\cdot)$, $\hat{P}(a;a)$, but $\neg\hat{P}(a)$.

There are renewal properties that are not liveness properties (e.g., the property that is only satisfied by the empty sequence), and there are liveness properties that

are not renewal properties (e.g., the nontermination property only satisfied by infinite executions). Some renewal properties, such as the one only satisfied by the empty sequence and the sequence $a; a$, are neither safety nor liveness. Although Alpern and Schneider [6] showed that exactly one property is both safety and liveness (the property satisfied by every execution), some interesting liveness properties are also renewal properties. We examine examples of such renewal properties in the following subsection.

3.3.2 Example Properties

We next present several examples of renewal properties that are not safety properties, as well as some examples of non-renewal properties. By the theorems in Sections 3.1.2 and 3.2.2, the non-safety renewal properties are effectively$_=$ enforceable by edit automata but not by truncation automata. Moreover, the proof of Theorem 8 shows how to construct an edit automaton to enforce any of the renewal properties described in this subsection. Later, in Section 4.4.3, we examine additional non-safety properties and show how they can be specified and enforced using practical program monitors.

Renewal properties Suppose we wish to constrain a user's interaction with a computer system. A user may execute any sequence of actions that does not involve opening files but must eventually log out. The process of executing non-file-open actions and then logging out may repeat indefinitely, so we might write the requisite property $\hat{P}$ more specifically as $(a_1{}^\star; a_2)^\infty$, where a_2 ranges over all actions for logging out, a_3 over actions for opening files, and a_1 over all other actions.[2] This $\hat{P}$

[2]As Alpern and Schneider note [6], this sort of $\hat{P}$ might be expressed with the (strong) *until* operator in temporal logic; event a_1 occurs *until* event a_2.

is not a safety property because a finite sequence of only a_1 events disobeys $\hat{P}$ but can be extended (by appending a_2) to obey $\hat{P}$. Our $\hat{P}$ is also not a liveness property because there are finite executions that cannot be extended to satisfy $\hat{P}$, such as the sequence containing only a_3. However, this non-safety, non-liveness property is a renewal property because infinite-length executions are valid if and only if they contain infinitely many (valid) prefixes of the form $(a_1{}^\star; a_2)^\star$.

Interestingly, if we enforce the policy described above on a system that only has actions a_1 and a_2, we remove the safety aspect of the property to obtain a liveness property that is also a renewal property. On the system $\{a_1, a_2\}$, the property satisfied by any execution matching $(a_1{}^\star; a_2)^\infty$ is a liveness property because any illegal finite execution can be made legal by appending a_2. The property is still a renewal property because an infinite execution is invalid if and only if it contains a finite number of valid prefixes after which a_2 never appears.

There are other interesting properties that are both liveness and renewal. For example, consider a property $\hat{P}$ specifying that an execution that does anything must eventually perform an audit by executing some action a. This is similar to the example renewal property given in Section 3.2.2. Because we can extend any invalid finite execution with the audit action to make it valid, $\hat{P}$ is a liveness property. It is also a renewal property because an infinite-length valid execution must have infinitely many prefixes in which a appears, and an infinite-length invalid execution has no valid prefix (except the empty sequence) because a never appears. Note that for this "eventually audits" renewal property to be enforceable by an edit automaton, we have to consider the empty sequence valid.

As briefly mentioned in Section 3.2.2, edit automata derive their power from being able to operate in a way similar to intentions files in database transactions. At

a high level, any transaction-based property is a renewal property. Let τ range over finite sequences of single, valid transactions. A transaction-based policy could then be written as τ^∞; a valid execution is one containing any number of valid transactions. Such transactional properties can be non-safety because executions may be invalid within a transaction but become valid at the conclusion of that transaction. Transactional properties can also be non-liveness when there exists a way to irremediably corrupt a transaction (e.g., every transaction beginning with *start*;*self-destruct* is illegal). Nonetheless, transactional properties are renewal properties because infinite-length executions are valid if and only if they contain an infinite number of prefixes that are valid sequences of transactions. The renewal properties described above as matching sequences of the form $(a_1{}^\star; a_2)^\infty$ can also be viewed as transactional; each transaction must match $a_1{}^\star; a_2$.

Non-renewal properties An example of an interesting liveness property that is not a renewal property is general availability. Suppose that we have a system with actions o_i for opening (or acquiring) and c_i for closing (or releasing) some resource i. Our policy $\hat{P}$ is that for all resources i, if i is opened, it must eventually be closed. This is a liveness property because any invalid finite sequence can be made valid simply by appending actions to close every open resource. However, $\hat{P}$ is not a renewal property because there are valid infinite sequences, such as $o_1; o_2; c_1; o_3; c_2; o_4; c_3; ...$, that do not have an infinite number of valid prefixes. An edit automaton can only enforce this sort of availability property when the number of resources is limited to one (in this case, the property is transactional: valid transactions begin with o_1 and end with c_1). Even on a system with two resources, infinite sequences like $o_1; o_2; c_1; o_1; c_2; o_2; c_1; o_1; ...$ prevent this resource-availability property from being a

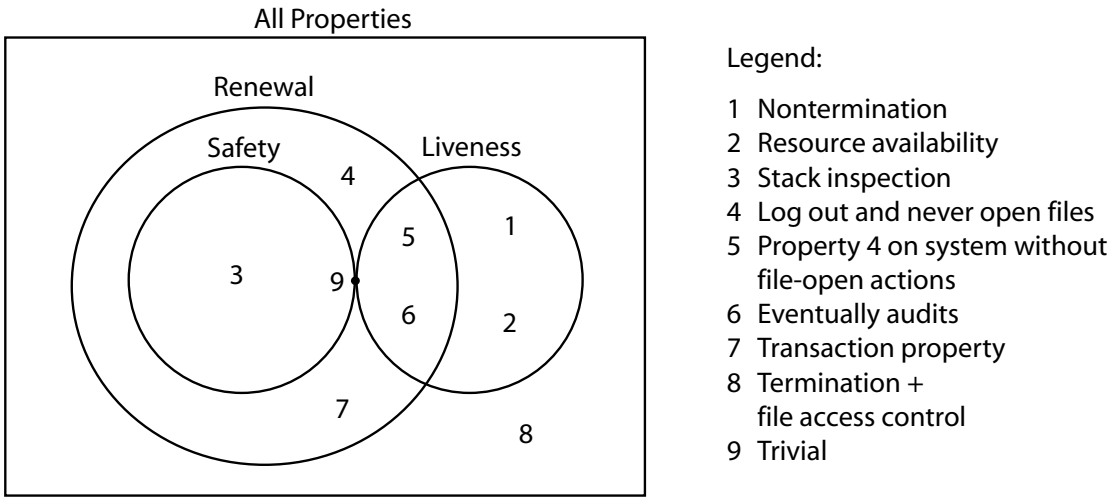

Figure 3.1: Relationships between safety, liveness, and renewal properties.

renewal property. Please note, however, that we have been assuming effective$_=$ enforcement; in practice we might find that $o_1; o_2; c_1 \cong o_1; c_1; o_2$, in which case edit automata *can* effectively$_\cong$ enforce these sorts of availability properties.

Of course, there are many non-renewal, non-liveness properties as well. We can arrive at such properties by combining a safety property with any property that is a liveness but not a renewal property. For example, termination is not a renewal property because invalid infinite sequences have an infinite number of valid prefixes. Termination is however a liveness property because any finite execution is valid. When we combine this liveness, non-renewal property with a safety property, such as that no accesses are made to private files, we arrive at the non-liveness, non-renewal property in which executions are valid if and only if they terminate and never access private files. The requirement of termination prevents this from being a renewal property; moreover, this property is outside the upper bound of what is effectively$_=$ enforceable by edit automata.

Figure 3.1 summarizes the results of the preceding discussion and that of Section 3.3.1. The Trivial property in Figure 3.1 considers all executions legal and is the only property in the intersection of safety and liveness properties.

Chapter 4

Enforcing Policies with Polymer

In the previous chapter, we found that program monitors can enforce a rich set of security policies. However, due to the problem of policy complexity (discussed in Chapter 1), actually specifying the policies we need to enforce is often difficult in practice. We propose handling policy complexity with a new language and system called Polymer that allows security engineers to create and enforce complex run-time policies on Java applications.

4.1 Polymer's Approach to Policy Complexity

In Polymer, security policies are first-class objects structured to be arbitrarily composed with other policies. This allows us to specify complex policies as compositions of much simpler policy modules.

Polymer programmers implement security policies by extending Polymer's `Policy` class, which is given a special interpretation by the underlying run-time system. Intuitively, each `Policy` object contains three main elements.

1. An effect-free decision procedure that determines how to react to security-sensitive application *actions*, which are method calls that monitors intercept. The `action` objects to which policies react contain information about the security-sensitive method's name, signature, calling object (if any), and actual arguments (if any).

2. Security state, which can be used to keep track of the application's activity during execution.

3. Methods to update the policy's security state.

We call the decision procedure mentioned above a *query* method. This method returns one of six *suggestions* indicating that: the action is *irrelevant* to the policy; the action is *OK* but relevant; the action should be reconsidered after some other code is *inserted*; the return value of the action should be *replaced* by a precomputed value (which may have been computed using earlier insertion suggestions); a security *exception* should be thrown instead of executing the action; or, the application should be *halted*. These objects are referred to as suggestions because there is no guarantee that the policy's desired reaction will occur when it is composed with other policies. Also for this reason, the query method should not have effects. State updates occur in other policy methods, which are invoked only when a policy's suggestion is followed.

In order to further support flexible but modular security policy programming, we treat all policies, suggestions, and application actions as first-class objects. Consequently, it is possible to define higher-order security policies that query one or more subordinate policies for their suggestions and then combine these suggestions in a semantically meaningful way, returning the overall result to the system, or other

policies higher in the hierarchy. We facilitate programming with suggestions and application events by introducing pattern-matching facilities and mechanisms that allow programmers to summarize a collection of application events as an *abstract action*.

4.2 Polymer System Overview

Similarly to the designs of Naccio [22] and PoET/Pslang [20], the Polymer system is composed of two main tools. The first is a policy compiler (implemented by Lujo Bauer) that compiles program monitors defined in the Polymer language into plain Java and then into Java bytecode. The second tool is a bytecode rewriter that processes ordinary Java bytecode, inserting calls to the monitor in all the necessary places. In order to construct a secure executable using these tools, programmers perform the following six steps.

1. Write the *action declaration file*, which lists all program methods that might have an impact on system security.

2. Instrument the system libraries specified in the action declaration file using the bytecode rewriter. This step may be performed independently of the specification of the security policy. The libraries must be instrumented before the Java Virtual Machine (JVM) starts up since the default JVM security constraints prevent many libraries from being modified or reloaded once the JVM is running.

3. Write and compile the security policy. The policy compiler translates the Polymer policy into ordinary Java and then invokes a Java compiler to trans-

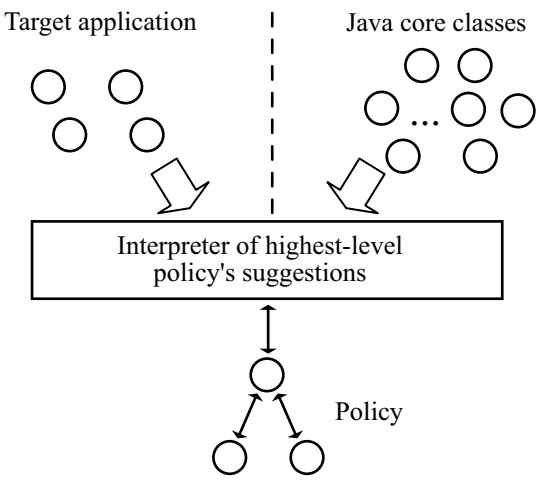

Figure 4.1: A secure Polymer application

late it to bytecode. Polymer's policy language is described in Section 4.3; its formal semantics appear in Chapter 5.

4. Start the JVM with the modified libraries.

5. Load the target application. During this loading, our specialized class loader rewrites the target code in the same way we rewrote the libraries in step two.

6. Execute the secured application.

Figure 4.1 shows the end result of the process. The instrumented target and library code run inside the JVM. Whenever this code is about to invoke a security-sensitive method, control is redirected through a generic policy manager, which queries the current policy. The current policy will return a suggestion that is interpreted by the policy manager.

```
public class Action {
  //'caller' is the object on which this action is invoked,
  //'fullSig' is the action's full signature, and
  //'params' are the action's actual arguments
  public Action(Object caller, String fullSig, Object[ ] params)
  public boolean equals(Action a)
  public Object[ ] getParams()
  public Class[ ] getParamClasses()
  public Object getCaller()
  public String getMethodName()
  public String getPackageName()
  public String getClassName()
  public String getSignature()
  public String toString()
}
```

Figure 4.2: Basic Polymer API for `Action` objects

4.3 The Polymer Language

In this section, we describe the core features of the Polymer language. We begin with the basic concepts and show how to program simple policies. Then, we demonstrate how to create more complex policies by composing simpler ones.

4.3.1 Core Concepts

Polymer is based on three central abstractions: actions, suggestions, and policies. Policies analyze actions and convey their decisions by means of suggestions.

Actions Monitors intercept and reason about how to react to security-sensitive method invocations. `Action` objects contain all of the information relevant to such invocations: static information such as the method signature, and dynamic information like the calling object and the method's parameters. The basic `Action` API is shown in Figure 4.2.

For convenient manipulation of actions, Polymer allows them to be matched against *action patterns*. An `Action` object matches an action pattern when the action's signature matches the one specified in the pattern. Patterns can use wildcards: `*` matches any one constraint (e.g., any return type or any single parameter type), and `..` matches zero or more parameter types. For example, the pattern

```
<public void java.io.*.<init>(int, ..)>
```

matches all public constructors in all classes in the `java.io` package whose first parameter is an `int`. In place of ⟨`init`⟩, which refers to a constructor, we could have used an identifier that refers to a particular method.

Action patterns appear in two places. First, the action declaration file is a set of action patterns. During the instrumentation process, every action that matches an action pattern in the action declaration file is instrumented. Second, policies use action patterns in `aswitch` statements to determine which security-sensitive action they are dealing with. `aswitch` statements are similar to Java's `switch` statements, as the following example shows.

```
aswitch(a) {
  case <void System.exit(int status)>: E;
  ...
}
```

If `Action a` represents an invocation of `System.exit`, this statement evaluates expression `E` with the variable `status` bound to the value of the method's single parameter.

Suggestions Whenever the untrusted application attempts to execute a security-relevant action, the monitor suggests a way to handle this action (which we often call a *trigger action* because it triggers the monitor into making such a suggestion).

The monitor conveys its decision about a particular trigger action using a `Sug` object. Polymer supplies a concrete subclass of the abstract `Sug` class for each type of suggestion mentioned in Section 4.1:

- An `IrrSug` suggests that the trigger action execute unconditionally because the policy does not reason about it.
- An `OKSug` suggests that the trigger action execute even though the action is of interest to the policy.
- An `InsSug` suggests that making a final decision about the target action be deferred until after some auxiliary code is executed and its effects are evaluated.
- A `ReplSug` suggests replacing the trigger action, which computes some return value, with a return value supplied by the policy. The policy may use `InsSug`s to compute the suggested return value.
- An `ExnSug` suggests that the trigger action not be allowed to execute, but also that the target be allowed to continue running. Whenever following an `ExnSug`, Polymer notifies the target that its attempt at invoking the trigger action has been denied by throwing a `SecurityException` that the target can catch before continuing execution.
- A `HaltSug` suggests that the trigger action not be allowed to execute and that the target be halted.

Breaking down the possible interventions of monitors into these categories provides great flexibility. In addition, this breakdown, which was refined by experience with writing security policies in Polymer, simplifies our job tremendously when it comes to controlling monitor effects and building combinators that compose monitors in sensible ways (see Section 4.3.3).

The reader might wonder why we distinguish between *irrelevant* and *OK* suggestions; there are two reasons. First, for performance, we need not update policy state when a policy considers the current action irrelevant. Second, some interesting superpolicies, such as the `Dominates` combinator described in Section 4.3.3, make a semantic distinction between subpolicies' `IrrSug` and `OKSug` suggestions.

Figure 4.3 shows the full contents of our `Sug` class. It contains convenience methods for dynamically determining a suggestion's concrete type, as well as methods for obtaining the policy that made the suggestion, the action that triggered that policy to make the suggestion, and any other suggestions and auxiliary objects the suggestion's creator found convenient to store in the `Sug`. Thus, when given a `Sug` object, Polymer policies can determine the precise circumstances under which that suggestion was made.

Policies Programmers encode a run-time monitor in Polymer by extending the base `Policy` class (Figure 4.4). A new policy must provide an implementation of the `query` method and may optionally override the `accept` and `result` methods.

- `query` analyzes a trigger action and returns a suggestion indicating how to deal with it.

```
public abstract class Sug {
  public abstract boolean isIrr();
  public abstract boolean isOK();
  public abstract boolean isRepl();
  public abstract boolean isExn();
  public abstract boolean isHalt();
  public abstract boolean isInsertion();
  public abstract Policy getSuggestingPolicy();
  public abstract Action getTrigger();
  public abstract Suggestion[ ] getSuggestions();
  public abstract Object getAuxiliaryObject();
}
```

Figure 4.3: Polymer's abstract `Sug` class

```
public abstract class Policy {
  public abstract Sug query(Action a);
  public void accept(Sug s) { };
  public void result(Sug s, Object result, boolean wasExnThn) { };
}
```

Figure 4.4: The parent class of all Polymer policies

- `accept` is called to indicate to a policy that its suggestion is about to be followed. This gives the policy a chance to perform any bookkeeping needed before the suggestion is carried out.

- `result` gives the policy access to the return value produced by following its `InsSug` or `OKSug`. The three arguments to `result` are the original suggestion the policy returned, the return value of the trigger action or inserted action (`null` if the return type was `void` and an `Exception` value if the action completed abnormally), and a flag indicating whether the action completed abnormally.

The `accept` method is called before following any suggestion except an `IrrSug`; the `result` method is only called after following an `OKSug` or `InsSug`. After `result`

```
public class Trivial extends Policy {
  public Sug query(Action a) { return new IrrSug(this); }
}
```

Figure 4.5: Polymer policy that allows all actions

is called with the result of an `InsSug`, the policy is queried again with the original trigger action (in response to which the policy had just suggested an `InsSug`). Thus, `InsSug`s allow a policy to delay making a decision about a trigger action until after executing another action.

A policy interface consisting of `query`, `accept`, and `result` methods is fundamental to the design of Polymer. We can compose policies by writing policy combinators that `query` other policies and combine their suggestions. In combining suggestions, a combinator may choose not to follow the suggestions of some of the queried policies. Thus, `query` methods must not assume that their suggestions will be followed and should be free of effects such as state updates and I/O operations.

4.3.2 Simple Policies

To give a feel for how to write Polymer policies, we define several simple examples in this section; in Sections 4.3.3 and 4.4.2 we will build more powerful policies by composing the basic policies presented here using a collection of policy combinators.

We begin by considering the most permissive policy possible: one that allows everything. The Polymer code for this policy is shown in Figure 4.5. Because the `query` method of `Trivial` always returns an `IrrSug`, it allows all trigger actions to execute unconditionally. To enable convenient processing of suggestions, every `Sug` constructor has at least one argument, the `Policy` making the `Sug`.

```
public class DisSysCalls extends Policy {
   public Sug query(Action a) {
     aswitch(a) {
       case <* java.lang.Runtime.exec(..)>:
         return new HaltSug(this, a);
     }
     return new IrrSug(this);
   }
   public void accept(Sug s) {
     if(s.isHalt()) {
       System.err.println("Illegal exec method called");
       System.err.println("About to halt target");
     }
   }
}
```

Figure 4.6: Polymer policy that disallows `Runtime.exec` methods

For our second example, we consider a more useful policy that disallows executing external code, such as OS system calls, via `java.lang.Runtime.exec(..)` methods. This policy, shown in Figure 4.6, simply halts the target when it calls `java.lang.Runtime.exec`. The `accept` method notifies the user of the security violation. Notice that this notification does not appear in the `query` method because it is an effectful computation; the notification should not occur if the policy's suggestion is not followed.

In practice, there can be many methods that correspond to a single action that a policy considers security relevant. For example, a policy that logs incoming email may need to observe all actions that can open a message. It can be cumbersome and redundant to have to enumerate all these methods in a policy, so Polymer makes it possible to group them into *abstract actions*. Abstract actions allow a policy to reason about security-relevant actions at a different level of granularity than is offered by the Java core API. They permit policies to focus on regulating

particular behaviors, say, opening email, rather than forcing them to individually regulate each of the actions that cause this behavior. This makes it easier to write more concise, modular policies. Abstract actions also make it possible to write platform-independent policies. For example, the set of actions that fetch email may not be the same on every system, but as long as the implementation of the abstract `GetMail` action is adjusted accordingly, the same policy for regulating email access can be used everywhere.

Figure 4.7 shows an abstract action for fetching email messages. The `matches` method of an abstract action returns `true` when a provided concrete action is one of the abstract action's constituents. The method has access to the concrete action's run-time parameters and can use this information in making its decision. All constituent concrete actions may not have the same parameter and return types, so one of the abstract action's tasks is to export a consistent interface to policies. This is accomplished via `convertParameter` and `convertResult` methods. The `convertResult` method in Figure 4.7 allows the `GetMail` abstract action to export a return type of `Message[ ]`.

Naccio [22] implements an alternative notion, called platform interfaces, that supports a similar sort of separation between concrete and abstract actions. It appears that our design is slightly more general, as our abstract actions allow programmers to define many-many relationships, rather than many-one relationships, between concrete and abstract actions. In addition, our abstract actions are first-class objects that may be passed to and from procedures, and we support the convenience of general-purpose pattern matching.

The example policy in Figure 4.8 logs all incoming email and prepends the string "SPAM:" to subject lines of messages flagged by a spam filter. To log incoming

```
public class GetMail extends AbsAction {
  public boolean matches(Action a) {
    aswitch(a) {
      case <Message IMAPFolder.getMessage(int)> :
      case <void IMAPFolder.fetch(Message[ ], *)> :
      case <Message IMAPFolder.getMessageByUID(long)> :
      case <Message[ ] IMAPFolder.getMessagesByUID(long[ ])> :
      case <Message[ ] IMAPFolder.getMessagesByUID(long, long)> :
      case <Message[ ] IMAPFolder.search(..)> :
      case <Message[ ] IMAPFolder.expunge()>:
      case <Message[ ] POP3Folder.expunge()>:
      case <void POP3Folder.fetch(Message[ ], *)>:
      case <Message POP3Folder.getMessage(int)>:
        return true;
    }
    return false;
  }
  public static Object convertResult(Action a, Object res) {
    aswitch(a) {
      case <Message IMAPFolder.getMessage(int)> :
      case <Message IMAPFolder.getMessageByUID(long)> :
        return new Message[ ] {(Message)res};
      case <void IMAPFolder.fetch(Message[ ] ma, *)> :
        return ma;
      case <void POP3Folder.fetch(javax.mail.Message[ ] ma, *)>:
        return ma;
      case <Message POP3Folder.getMessage(int)>:
        return new Message[ ] {(Message)res};
      default:
        return res;
    }
  }
}
```

Figure 4.7: Abstract action for receiving email messages; the action's signature is `Message[ ] GetMail()`

```
public class IncomingMail extends Policy {
  ...
  public Sug query(Action a) {
    aswitch(a) {
      case <abs * examples.mail.GetMail()>:
        return new OKSug(this, a);
      case <* MimeMessage.getSubject()>:
      case <* IMAPMessage.getSubject()>:
        String subj = spamifySubject(a.getCaller());
        return new ReplSug(this, a, subj);
      case <done>:
        if(!isClosed(logFile))
          return new InsSug(this, a, new Action(logFile,
            "java.io.PrintStream.close()", new Object[ ]{}));
    }
    return new IrrSug(this, a);
  }
  public void result(Sug sugg, Object res, boolean wasExnThn) {
    if(!sugg.isOK() || wasExnThn) return;
    log(GetMail.convertResult(sugg.getTrigger(), result));
  }
}
```

Figure 4.8: Abbreviated Polymer policy that logs all incoming email and prepends the string "SPAM:" to subject lines on messages flagged by a spam filter

mail, the policy first tests whether the trigger action matches the `GetMail` abstract action (from Figure 4.7), using the keyword `abs` in an action pattern to indicate that `GetMail` is abstract. Since `query` methods should not have effects, the policy returns an `OKSug` for each `GetMail` action; the policy logs the fetched messages in the `result` method. Polymer triggers a `done` action when the application terminates; the policy takes advantage of this feature to insert an action that closes the message log. If the `InsSug` recommending that the log be closed is accepted, the policy will be queried again with a `done` action after the inserted action has been executed. In the second query, the log file will already be closed, so the policy will return an `IrrSug`. The

policy also intercepts calls to `getSubject` in order to mark email as spam. Instead of allowing the original call to execute, the policy fetches the original subject, prepends "SPAM:" if necessary, and returns the result via a `ReplSug`. Running a spam filter on the client side allows an end user to filter based on individually customized rules.

Sometimes, a policy requires notifying the target that executing its trigger action would be a security violation. When no suitable return value can indicate this condition to the target, the policy may make an `ExnSug` rather than a `ReplSug`. For example, the email `Attachments` policy in Figure 4.9 seeks user confirmation before creating an executable file. When the user fails to provide the confirmation, the `Attachments` policy signals a policy violation by returning an `ExnSug`, rather than by halting the target outright. This `ExnSug` causes a `SecurityException` to be thrown, which can be caught by the application and dealt with in an application-specific manner.

4.3.3 Policy Combinators

Polymer supports policy modularity and code reuse by allowing policies to be combined with and modified by other policies. In Polymer, a policy is a first-class Java object, so it may serve as an argument to and be returned by other policies. We call a policy parameterized by other policies a *policy combinator*. When referring to a complex policy with many policy parts, we call the policy parts *subpolicies* and the complex policy a *superpolicy*.

We have written and provide with Polymer a library of common combinators [12]; however, policy architects are always free to develop new combinators to suit their own specific needs. Studying which combinators are the "right" ones to have available, for various definitions of "right" (e.g., that are the minimal necessary to specify

```
import javax.swing.*;
public class Attachments extends Policy {
  private boolean isNameBad(String fn) {
    return(fn.endsWith(".exe") || fn.endsWith(".vbs") ||
           fn.endsWith(".hta") || fn.endsWith(".mdb"));
  }
  private boolean userCancel = false; //user disallowed the file?
  private boolean noAsk = false; //can we skip the confirmation?
  public Sug query(polymer.Action a) {
    aswitch(a) {
      case <abs void examples.mail.FileWrite(String fn)>:
        if(noAsk) return new OKSug(this, a);
        if(userCancel) return new ExnSug(this, a);
        polymer.Action insAct = new polymer.Action(null,
          "javax.swing.JOptionPane.showConfirmDialog(" +
            "java.awt.Component, java.lang.Object, " +
            "java.lang.String, int)",
          new Object[ ]{null, "The target is creating file: " + fn +
              " This is a dangerous file type. " +
              "Are you sure you want to create this file?",
            "Security Question",
            new Integer(JOptionPane.YES_NO_OPTION)});
        if(isNameBad(fn)) return new InsSug(this, a, insAct);
        return new IrrSug(this, a);
    }
    return new IrrSug(this);
  }
  public void accept(Sug s) {
    if(s.isExn()) userCancel = false;
    if(s.isOK()) noAsk = false;
  }
  public void result(Sug s, Object result, boolean wasExnThn) {
    if(s.isInsertion() && ((Integer)result).intValue() ==
      JOptionPane.NO_OPTION) userCancel = true;
    if(s.isInsertion() && ((Integer)result).intValue() ==
      JOptionPane.YES_OPTION) noAsk = true;
  }
}
```

Figure 4.9: Polymer policy that seeks confirmation before creating `.exe`, `.vbs`, `.hta`, and `.mdb` files

all compositions of stack-inspection policies, that are guaranteed to terminate, or that satisfy interesting properties such as associativity, commutativity, and distributivity), would make interesting future work. Because the definition of which combinators are "right" can vary from application to application, it is important to have a framework like Polymer in which arbitrary combinators can be expressed.

We next describe several types of combinators we have developed and found useful in practice. The email policy described in Section 4.4.2 includes all of them. Although our combinators were developed through experience and seem to match intuitive notions of policy conjunction, precedence, etc., we have not formalized their semantics beyond the intuition given in this subsection and their Polymer code implementations. Nonetheless, Krishnan provides formal semantics for several of our combinators [34].

Conjunctive combinator It is often useful to restrict an application's behavior by applying several policies at once and, for any particular trigger action, enforcing the most restrictive one. For example, a policy that disallows access to files can be used in combination with a policy that disallows access to the network; the resulting policy disallows access to both files and the network. In the general case, the policies being conjoined may reason about overlapping sets of actions. When this is the case, we must consider what to do when the two subpolicies suggest different courses of action. In addition, we must define the order in which effectful computations are performed.

Our conjunctive combinator composes exactly two policies; we can generalize this to any number of subpolicies. Our combinator operates as follows.

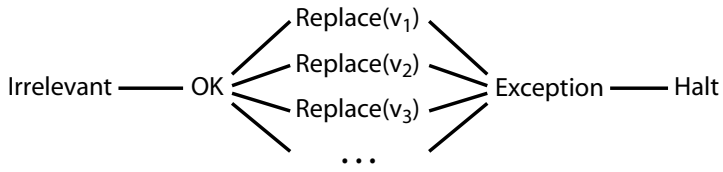

Figure 4.10: Lattice ordering of Polymer suggestions' semantic impact

- If either subpolicy suggests insertions, so does the combinator, with any insertions by the left (first) conjunct occurring prior to insertions by the right conjunct. Following the principle of complete mediation, the monitor will recursively examine these inserted actions if they are security-relevant.

- If neither subpolicy suggests insertions, the conjunctive combinator computes and returns the least upper bound of the two suggestions, as described by the lattice in Figure 4.10, which orders suggestions in terms of increasing semantic impact. For instance, `IrrSug` has less impact than `OKSug` since an `IrrSug` indicates the current method is allowed but irrelevant to the policy whereas `OKSug` says it is allowed, but relevant, and updates of security state may be needed. `ReplSug`s have more impact than `OKSug`s since they change the semantics of the application. `ReplSug`s containing different replacements are considered inequivalent; consequently, the "conjunction" of two `ReplSug`s is considered to be an `ExnSug`.

Note that a sequence of insertions made by one conjunct may affect the second conjunct. In fact, this is quite likely if the second conjunct considers the inserted actions security-relevant. In this case, the second conjunct may make a different suggestion regarding how to handle an action before the insertions than it does afterward. For example, in the initial state the action might have been OK, but after

the intervening insertions the second conjunct might suggest that the application be halted.

Figure 4.11 contains our conjunctive combinator. The invocations of `SugUtils.cpSug` in the query method simply create new suggestions with the same type as the first parameter in these calls. Notice that the suggestion returned by the combinator includes the suggestions on which the combinator based its decision. This design makes it possible for the combinator's `accept` and `result` methods to notify the appropriate subpolicies that their suggestions have been accepted and followed.

Precedence combinators We have found the conjunctive policy to be the most common combinator. However, it is useful on occasion to have a combinator that gives precedence to one subpolicy over another. One example is the `TryWith` combinator (shown in Figure 4.12), which queries its first subpolicy, and if that subpolicy returns an `IrrSug`, `OKSug`, or `InsSug`, it makes the same suggestion. Otherwise, the combinator defers judgment to the second subpolicy. The email policy described in Section 4.4.2 uses the `TryWith` combinator to join a policy that allows only HTTP connections with a policy that allows only POP and IMAP connections; the resulting policy allows exactly those kinds of connections and no others.

A similar sort of combinator is the `Dominates` combinator, which always follows the suggestion of the first conjunct if that conjunct considers the trigger action security-relevant; otherwise, it follows the suggestion of the second conjunct. Note that if two subpolicies never consider the same action security-relevant, composing them with a `Dominates` combinator is equivalent to composing them with a `Conjunction` combinator, except the `Dominates` combinator is in general more ef-

```
public class Conjunction extends Policy {
  private Policy p1, p2; //subpolicies
  public Conjunction(Policy p1, Policy p2) {this.p1=p1; this.p2=p2;}

  public Sug query(Action a) {
    //return the most restrictive subpolicy suggestion
    Sug s1=p1.query(a), s2=p2.query(a);
    if(SugUtils.areSugsEqual(s1,s2))
      return SugUtils.cpSug(s1, this, a, new Sug[ ]{s1,s2});
    if(s1.isInsertion())
      return SugUtils.cpSug(s1, this, a, new Sug[ ]{s1});
    if(s2.isInsertion())
      return SugUtils.cpSug(s2, this, a, new Sug[ ]{s2});
    if(s1.isHalt()) return SugUtils.cpSug(s1,this,a,new Sug[ ]{s1});
    if(s2.isHalt()) return SugUtils.cpSug(s2,this,a,new Sug[ ]{s2});
    if(s1.isExn()) return SugUtils.cpSug(s1,this,a,new Sug[ ]{s1});
    if(s2.isExn()) return SugUtils.cpSug(s2,this,a,new Sug[ ]{s2});
    if(s1.isRepl() && s2.isRepl()) return new ExnSug(this, a);
    if(s1.isRepl()) return SugUtils.cpSug(s1,this,a,new Sug[ ]{s1});
    if(s2.isRepl()) return SugUtils.cpSug(s2,this,a,new Sug[ ]{s2});
    if(s1.isOK()) return SugUtils.cpSug(s1,this,a,new Sug[ ]{s1});
    if(s2.isOK()) return SugUtils.cpSug(s2,this,a,new Sug[ ]{s2});
    return new IrrSug(this, a);
  }
  public void accept(Sug sug) {
    //notify subpolicies whose suggestions were accepted
    Sug[ ] sa = sug.getSuggestions();
    for(int i = 0; i < sa.length; i++) {
      sa[i].getSuggestingPolicy().accept(sa[i]);
    }
  }
  public void result(Sug sug, Object result, boolean wasExnThn) {
    //notify subpolicies whose suggestions were followed
    Sug[ ] sa = sug.getSuggestions();
    for(int i = 0; i < sa.length; i++) {
      sa[i].getSuggestingPolicy().result(sa[i], result, wasExnThn);
    }
  }
}
```

Figure 4.11: A conjunctive policy combinator

```
public class TryWith extends Policy {
  //subpolicies
  private Policy p1, p2;

  public TryWith(Policy p1, Policy p2) {
    this.p1=p1;
    this.p2=p2;
  }

  public Sug query(Action a) {
    Sug s1=p1.query(a);

    //if p1 accepts or inserts, return its suggestion
    if(s1.isInsertion() || s1.isOK() || s1.isIrr())
      return SugUtils.cpSug(s1, this, a, new Sug[ ]{s1});

    //otherwise return whatever p2 suggests
    Sug s2=p2.query(a);
    return SugUtils.cpSug(s2, this, a, new Sug[ ]{s2});
  }

  public void accept(Sug sug) {
    //notify the subpolicy that made the now accepted suggestion
    Sug[ ] sa = sug.getSuggestions();
    sa[0].getSuggestingPolicy().accept(sa[0]);
  }

  public void result(Sug sug, Object result, boolean wasExnThn) {
    //notify the subpolicy that made the now followed suggestion
    Sug[ ] sa = sug.getSuggestions();
    sa[0].getSuggestingPolicy().result(sa[0], result, wasExnThn);
  }

}
```

Figure 4.12: The `TryWith` policy combinator

ficient because it need not always query both subpolicies. In our email policy we use `Dominates` to construct a policy that both restricts the kinds of network connections that may be established and prevents executable files from being created. Since these two subpolicies regulate disjoint set of actions, composing them with the `Conjunction` combinator would have needlessly caused the second subpolicy to be queried even when the trigger action was regulated by the first subpolicy, and therefore clearly not of interest to the second.

Selectors Selectors are combinators that choose to enforce exactly one of their subpolicies. The `IsClientSigned` selector of Section 4.4.2, for example, enforces a weaker policy on the target application if the target is cryptographically signed; otherwise, the selector enforces a stronger policy.

Policy modifiers Policy modifiers are higher-order policies that enforce a single policy while also performing some other actions. Suppose, for example, that we want to log the actions of a target application and the suggestions made by a policy acting on that target. Rather than modifying the existing policy, we can accomplish this by wrapping the policy in an `Audit` unary superpolicy. When queried, `Audit` blindly suggests whatever the original policy's `query` method suggests. `Audit`'s `accept` and `result` methods perform logging operations before invoking the `accept` and `result` methods of the original policy.

Another example of a policy modifier is our `AutoUpdate` superpolicy. This policy checks a remote site once per day to determine if a new policy patch is available. If so, it makes a secure connection to the remote site, downloads the updated policy, and dynamically loads the policy into the JVM as its new subpolicy. Policies of this sort, which determine how to update other policies at run time, are useful

because they allow new security constraints to be placed on target applications dynamically, as vulnerabilities are discovered. Note however that because library classes (such as `java.lang.Object`) cannot in general be reloaded while the JVM is running, policies loaded dynamically should consider security-relevant only actions appearing in the static action declaration file. For this reason, we encourage security programmers to be reasonably conservative when writing action declaration files for dynamically updateable policies.

A third useful sort of policy modifier is a `Filter` that blocks a policy from seeing certain actions. In some circumstances, self-monitoring policies can cause loops that will prevent the target program from continuing (for example, a policy might react to an action by inserting that same action, which the policy will then see and react to in the same way again). It is easy to write a `Filter` to prevent such loops. More generally, `Filter`s allow the superpolicy to determine whether an action is relevant to the subpolicy.

4.4 Empirical Evaluation

Experience implementing and using Polymer has been instrumental in confirming and refining our design.

4.4.1 Implementation

The principal requirement for enforcing the run-time policies in which we are interested is that the flow of control of a running program passes to a monitor immediately before and after executing a security-relevant method. The kind of pre- and post-invocation control-flow modifications to bytecode that we use to implement

Polymer can be done by tools like AspectJ [30]. Accordingly, we considered using AspectJ to insert into bytecode hooks that would trigger our monitor as needed. However, we wanted to retain precise control over how and where rewriting occurs to be able to make decisions in the best interests of security, which is not the primary focus of aspect-oriented languages like AspectJ. Instead, we used the Apache BCEL API [7] to develop our own bytecode rewriting tool.

Custom class loaders have often been used to modify bytecode before executing it [4, 8]; we use this technique also. Since libraries used internally by the JVM cannot be rewritten by a custom class loader, we rewrite those libraries before starting the JVM and the target application.

Performance It is instructive to examine the performance costs of enforcing policies using Polymer. We did not concentrate on making our implementation as efficient as possible, so there is much room for improvement here. However, the performance of our implementation does shed some light on the costs of run-time policy enforcement.

Our system impacts target applications in two phases: before and during loading, when the application and the class libraries are instrumented by the bytecode rewriter; and during execution. The total time to instrument every method in all of the standard Java library packages (i.e., the 28742 methods in the 3948 classes in the `java` and `javax` packages of Sun's Java API v.1.4.0), inserting monitor invocations at the proper times before and after every method executes, was 107 s, or 3.7 ms per instrumented method.[1] Because we only insert hooks for calling the

[1]The tests were performed on a Dell PowerEdge 2650 with dual Intel Xeon 2.2 GHz CPUs and 1 GB of RAM, running RedHat Linux 9.0. The times represent real time at low average load. We performed each test multiple times in sets of 100. The results shown are the average for the set with the lowest average, after removing outliers.

interpreter of the highest-level policy's suggestions (see Figure 4.1), rather than inlining or invoking any particular policy, we introduce a level of indirection that permits policies to be updated dynamically without rewriting application or library code. Hence, this instrumentation only needs to be performed once.

The average time to load non-library classes into the JVM with our specialized class loader, but without instrumenting any methods, was 12 ms, twice as long as the VM's default class loader required. The cost of transferring control to and from a Polymer policy while executing a target is very low (approximately 0.62 ms); the run-time overhead is dominated by the computations actually performed by the policy. Hence the cost of monitoring a program with Polymer is almost entirely dependent on the complexity of the security policy.

4.4.2 Case Study: Securing Email Clients

To test the usefulness of Polymer in practice, we have written a large-scale policy to secure untrusted email clients that use the JavaMail API. The entire policy, presented in Figure 4.13, is approximately 1800 lines of Polymer code. We have extensively tested the protections enforced by the policy on an email client called Pooka [45], without having to inspect or modify any of the approximately 50K lines of Pooka source code. The run-time cost of enforcing the complex constraints specified by our policy is difficult to measure because the performance of the email client depends largely on interactions with the user; however, our experience indicates that the overhead is rarely noticeable.

The component policies in Figure 4.13 each enforce a modular set of constraints. The `Trivial` and `Attachments` policies were described in Section 4.3.2; the `Conjunction`, `TryWith`, `Dominates`, `Audit`, and `AutoUpdate` superpolicies were de-

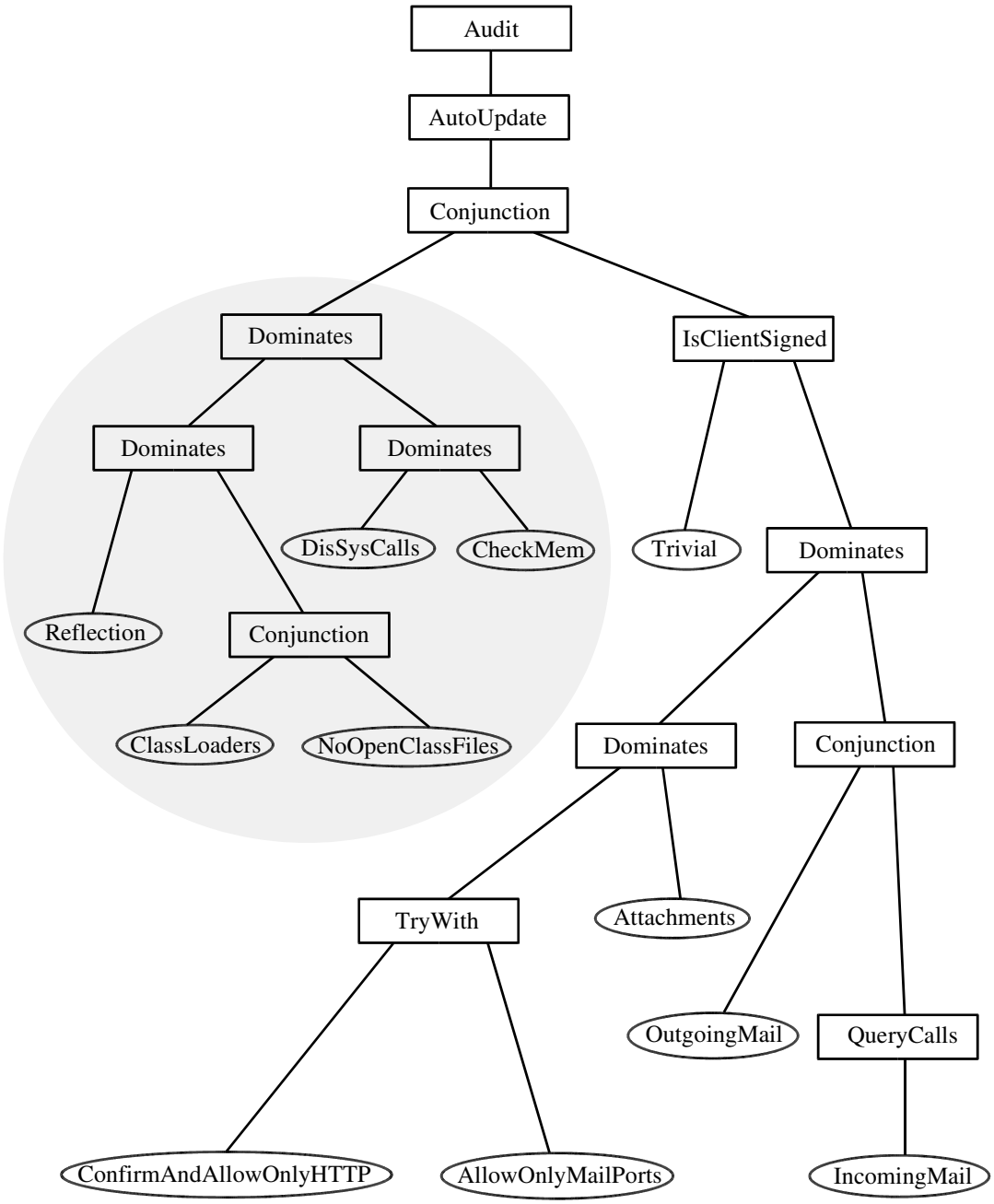

Figure 4.13: Email policy hierarchy

scribed in Section 4.3.3. The left branch of the policy hierarchy (shaded in Figure 4.13) describes a generic policy that we include in all of our high-level Polymer policies. This branch of policies ensures that a target cannot use class loading, reflection, or system calls maliciously and alerts the user when the memory available to the virtual machine is nearly exhausted (determined by generating interrupts to poll the `java.lang.Runtime.totalMemory` and `java.lang.Runtime.maxMemory` methods every four seconds). The nonshaded branch of the policy hierarchy describes policies specifically designed for securing an email client and enforces constraints as follows.

- `IsClientSigned` tests whether the email client is cryptographically signed. If it is, we run `Trivial` but continue to log security-relevant actions and allow dynamic policy updates. If the client is not signed, we run a more restrictive policy.

- `ConfirmAndAllowOnlyHTTP` pops up a window seeking confirmation before allowing HTTP connections, and disallows all other types of network connections.

- `AllowOnlyMailPorts` only allows socket connections to standard email ports (SMTP, regular POP and IMAP, and SSL-based POP and IMAP). This policy suggests throwing `SecurityException`s to prevent the target from making any other types of network connections. Figure 4.14 contains the code for this policy. Note that with a lot of engineering effort and run-time overhead, we could encode the low-level details of the email protocols into our policy to enforce a stronger `AllowOnlyMail` policy that would precisely ensure that untrusted email clients only make connections that obey standard email proto-

```
public class AllowOnlyMailPorts extends Policy {
  public Sug query(Action a) {
    aswitch(a) {
      case <abs void examples.mail.NetOpen(String addr, int port)>:
        String logStr = "Connection made to "+addr+", port "+port;
        if(port==143    //IMAP connection
          || port==993  //SSL IMAP connection
          || port==25   //SMTP connection
          || port==110  //POP3 connection
          || port==995) //SSL POP3 connection
          return new OKSug(this, a, null, logStr);
        else return new ExnSug(this, a);
    }
    return new IrrSug(this);
  }
  public void accept(Sug s) {
    if(s.isOK()) System.out.println(s.getAuxiliaryObject());
  }
}
```

Figure 4.14: Polymer policy that only allows network connections to email ports

cols. However, we have chosen to enforce the much simpler and lower-overhead `AllowOnlyMailPorts` policy that provides weaker, port-based guarantees.

- `QueryCalls` is a policy modifier that allows security-sensitive actions invoked in the `query` method of its subpolicy to execute unconditionally. `QueryCalls` OKs these actions without requerying the subpolicy in order to prevent infinite loops that can occur when the subpolicy invokes actions that it also monitors. The implementation of `QueryCalls` inspects the dynamic call stack to determine whether a trigger action was invoked in the subpolicy's `query` method.

- `OutgoingMail` logs all mail being sent, pops up a window confirming the recipients of messages (to prevent a malicious client from quietly sending mail

on the user's behalf), backs up every outgoing message by sending a BCC to `polydemo@cs.princeton.edu`, and automatically appends contact information to textual messages.

- `IncomingMail` was shown in an abbreviated form in Figure 4.8. In addition to logging incoming mail and prepending "SPAM:" to the subject lines of email that fails a spam filter, this policy truncates long subject lines and displays a warning when a message containing an attachment is opened.

4.4.3 Specifying Non-safety Policies in Polymer

Chapter 3 showed that program monitors modeled by edit automata can enforce some non-safety properties. Because Polymer implements all the ways in which edit automata can react to trigger actions—`InsSug` *inserts* an action, and `ReplSug`, `ExnSug`, and `HaltSug` are different ways to *suppress* actions (and `IrrSug` and `OKSug` are different ways to *accept* actions)—we can likewise use the Polymer system to enforce some non-safety properties. Although our case-study email policy is a safety property, for the remainder of this chapter we will study two reasonable and simple examples of non-safety policies enforceable in Polymer. The first example ensures that ATM machines generate a proper log when dispensing cash, while the second example ensures that targets writing to a file eventually give the file satisfactory contents.

ATM-logging Policy Let us first consider a simple ATM system for dispensing cash. It contains the following three methods.

1. `logBegin(`n`)` creates a log message that the ATM is about to dispense n dollars.

2. `dispense(`n`)` causes the ATM to dispense n dollars.

3. `logEnd(`n`)` creates a log message that the ATM just completed dispensing n dollars.

Suppose we wish to require that the ATM machine's software properly logs all cash dispensations; we will consider an execution valid if and only if it has the form $(\texttt{logBegin}(n); \texttt{dispense}(n); \texttt{logEnd}(n))^{\infty}$. That is, valid executions must be sequences of valid transactions, where each valid transaction consists of logging that some amount of cash is about to be dispensed, dispensing that cash, and then logging that that amount of cash has just been dispensed. Our desired policy is a non-safety, non-liveness, renewal property. It is non-safety because there exists an invalid execution ($\texttt{logBegin}(20)$) that prefixes a valid execution ($\texttt{logBegin}(20); \texttt{dispense}(20); \texttt{logEnd}(20)$). It is non-liveness because some invalid execution ($\texttt{dispense}(20)$) cannot be made valid through extension. Nonetheless, this non-safety, non-liveness property is clearly a transaction-style renewal property (as described in Section 3.3.2).

We can enforce this non-safety policy in Polymer simply by suppressing preliminary `logBegin` and `dispense` actions until we are guaranteed that the current transaction is valid, at which point the suppressed actions get re-inserted. Figure 4.15 contains the `AtmPolicy`, which employs this enforcement technique and suppresses actions simply by returning `ReplSug`s in the `query` method. For simplicity, we assume in the example code that all re-inserted methods complete normally. This frees us from worrying about exceptions being raised and VM-terminating actions being called during the inserted methods, which could prevent our policy from inserting all of the actions necessary to satisfy the property. We can remove this simplifying assumption by modifying our policy so that it catches exceptions

```
public class AtmPolicy extends Policy {
  //which dollar amount is the basis for the current transaction?
  private int amt = 0;
  //which state of the transaction are we in?
  //0=initial    1=logBegin    2=logBegin;dispense
  private int transState = 0;
  //are we in the process of inserting suppressed methods?
  private boolean isInsert = false;

  public Sug query(Action a) {
    aswitch(a) {
      case <void examples.ATM.logBegin(int n)>:
        if(transState==0) return new ReplSug(null, a);
        else return new HaltSug(a);
      case <void examples.ATM.dispense(int n)>:
        if(transState==1 && amt==n) return new ReplSug(null, a);
        else return new HaltSug(a);
      case <void examples.ATM.logEnd(int n)>:
        if(transState==2 && amt==n) return new OKSug(a);
        else return new HaltSug(a);
      default:
        if(transState>0) return new HaltSug(a);
        else return new IrrSug();
  }  }
  public void accept(Sug s) {
    aswitch(s.getTrigger()) {
      case <void examples.ATM.logBegin(int n)>:
        transState = 1; amt = n; break;
      case <void examples.ATM.dispense(int n)>:
        transState = 2;
    }
    if(s.isOK()) {   //transaction is valid
      isInsert = true;
      examples.ATM.logBegin(amt); examples.ATM.dispense(amt);
      isInsert = false;
      amt = 0; transState = 0;
  }  }
}
```

Figure 4.15: Non-safety Polymer policy ensuring that ATM cash dispensation gets logged properly

raised and `done` actions invoked during execution of inserted actions but does not allow the exceptions to propagate or the virtual machine to exit until all required insertions have been made.

File-contents Policy For another example, let us consider enforcing a property that allows files to be written, possibly using multiple file-write operations, if and only if the file contents eventually satisfy some predicate that is passed to the policy as a parameter. The predicate might not be satisfied in the middle of a sequence of writes but must be satisfied after a later write. For instance, the predicate might be true exactly when the file being written contains an appropriate ASCII copyright notice. This predicate must hold at the end of a sequence of file writes to ensure that every file on the system carries the proper copyright notice. This is not a safety property: we can overwrite a file's copyright notice with other data, making the execution invalid; however, we can extend the invalid execution to satisfy the property by appending the proper copyright text to the file. Actually, if we assume that the file predicate is satisfiable then this file-contents policy is a liveness property: any invalid finite execution becomes valid when the target executes whatever file-write operations satisfy the file predicate.

As expected, we enforce the non-safety, liveness, file-contents property by suppressing (feigning) writes to files until we can ensure their validity, at which point we re-insert all suppressed writes to the now-valid file. Figure 4.16 contains an abbreviated Polymer policy that enforces this file-contents property. In its constructor, `FilePredPolicy` accepts an object that implements the `FilePredicate` interface, which contains a method to test file validity when given a file name and a sequence of writes to that file. The `FilePredPolicy` uses the `FilePredicate` in its

```
public class FilePredPolicy extends Policy {

  //this interface contains a method for testing
  //whether a file's contents are OK
  private FilePredicate filePred;

  //store whether file writes are due to our inserting them
  private boolean areWeInserting = false;

  public FilePredPolicy(FilePredicate filePred) {
    this.filePred = filePred;
  }

  public Sug query(Action a) {
    if(areWeInserting)
      return new IrrSug(this);
    aswitch(a) {
      case <abs void absact.FileWrite(byte[ ] b, int off, int len)>:
        return new ReplSug(this, a, null);
      case <abs long absact.FileRead(byte[ ] b, int off, int len)>:
        return new ReplSug(this, a, readWithSuppressedWrites(a));
      default:
        return new IrrSug(this);
    }
  }

  public void accept(Sug s) {
    aswitch(s.getTrigger()) {
      case <abs void absact.FileWrite(byte[ ] b, int off, int len)>:
        //we are suppressing a write; next insert all suppressed
        //writes that need to be inserted, according to filePred
        areWeInserting = true;
        performInsertions(s.getTrigger());
        areWeInserting = false;
    }
  }

}
```

Figure 4.16: Abbreviated non-safety Polymer policy ensuring that files are eventually written satisfactorily

`performInsertions` method to check whether to insert a file's suppressed writes. The policy remembers which writes have been suppressed for each file by maintaining a mapping from files to ordered lists of pending write operations. Maintenance of this mapping occurs in the `performInsertions` method. In order to properly feign file write operations, `FilePredPolicy` also monitors actions that read data associated with files (i.e., file contents, lengths, and modification times), and ensures that the target sees files as if suppressed writes have actually executed. Hence, our Polymer monitor effectively enforces the file-contents policy: the monitor does not modify valid executions' semantics (though intermediate file writes may be performed later than they normally would), and the monitor ensures that all observed executions are valid.

Practicality Constraints This subsection has demonstrated that practical program monitors can sometimes enforce non-safety, and even liveness, renewal properties. The key reason we can enforce our example non-safety properties in practice is that monitors can successfully feign dangerous `logBegin`, `dispense`, and `FileWrite` actions. As described in Section 3.2.2, monitors in many situations lack the ability to feign, or even insert, the necessary actions, so there exist many renewal properties unenforceable by practical program monitors. In the future, we plan to refine our theoretical model to capture situations in which monitors lack the full computational abilities present in the executing machine.

Chapter 5

Formal Semantics of the Polymer Language

We next give a semantics to the core features of the Polymer language in order to precisely and unambiguously communicate the central workings of our language. We used Java as the basis for our Polymer implementation to make the system widely accessible and to take advantage of the wealth of existing Java libraries and applications. However, we choose to give the Polymer semantics in the context of a lambda calculus because lambda calculi are inherently simpler to specify than class-based languages such as Java (even the lightest-weight specification of Java such as Featherweight Java [27] is substantially more complex than the simply-typed lambda calculus). More importantly, the central elements of our policy language do not depend upon Java-specific features such as classes, methods, and inheritance. We could just as easily have implemented Polymer policies for a functional language such as ML [42] or a type-safe imperative language (type safety protects the program monitor's state and code from being tampered with by the untrusted application).

5.1 Syntax

Figure 5.1 describes the main syntactic elements of the calculus. The language is simply-typed with types for booleans, n-ary tuples, references, and functions. Our additions include simple base types for policies (Poly), suggestions (Sug), actions (Act), which are suspended function applications, and results of those suspended function applications (Res).

Programs as a whole are 4-tuples consisting of a collection of functions that may be monitored, a memory that maps memory locations to values, and two expressions. The first expression represents the security policy; the second expression represents the untrusted application. Execution of a program begins by reducing the policy expression to a policy value. It continues by executing the application expression in the presence of the policy.

Monitored functions ($\mathrm{fun} f(x{:}\tau_1){:}\tau_2\{e\}$) have global scope and are syntactically separated from ordinary functions ($\lambda x{:}\tau.e$).[1] Moreover, we treat monitored function names f, which have global scope and may therefore alpha-vary over entire programs, as a syntactically separate class of variables from ordinary variables x. Monitored function names are unique and may only appear wrapped up as actions as in $\mathrm{act}(f, e)$. These actions are suspended computations that must be explicitly *invoked* with the command invk e. Invoking an action causes the function in question to be executed and its result wrapped in a result constructor $\mathrm{result}(e{:}\tau)$. The elimination forms for results and most other objects discussed above is handled through a generic case expression and pattern matching facility. The class of patterns p includes variable patterns x as well as patterns for matching constructors.

[1] As usual, we treat expressions that differ only in the names of their bound variables as identical. We often write let $x = e_1$ in e_2 for $(\lambda x{:}\tau.e_2)e_1$.

types :

τ ::= Bool | $(\vec{\tau})$ | τ Ref | $\tau_1 \rightarrow \tau_2$ | Poly | Sug | Act | Res

programs :

P ::= $(\vec{F}, M, e_{\text{pol}}, e_{\text{app}})$

monitored functions :

F ::= fun$f(x{:}\tau_1){:}\tau_2\{e\}$

memories :

M ::= $\cdot$ | $M, l : v$

values :

v ::= true | false | $(\vec{v})$ | l | $\lambda x{:}\tau.e$ | pol$(v_{\text{query}}, v_{\text{acc}}, v_{\text{res}})$ | irrs | oks | inss(v) |
repls(v) | exns | halts | act(f, v) | result$(v{:}\tau)$

expressions :

e ::= v | x | $(\vec{e})$ | $e_1; e_2$ | ref e | !e | $e_1{:=}e_2$ | $e_1\ e_2$ | pol$(e_{\text{query}}, e_{\text{acc}}, e_{\text{res}})$ | inss(e) |
repls(e) | act(f, e) | invk e | result$(e{:}\tau)$ | case e_1 of $(p \Rightarrow e_2$ | _ $\Rightarrow e_3)$ |
try e_1 with e_2 | raise exn | abort

patterns :

p ::= x | true | false | $(\vec{p})$ | pol(x_1, x_2, x_3) | irrs | oks | inss(p) | repls(p) | exns |
halts | act(f, p) | result$(p{:}\tau)$

Figure 5.1: Formal syntax for the Polymer calculus

Ordinary, unmonitored functions are executed via the usual function application command ($e_1\ e_2$).

To create a policy, one applies the policy constructor pol to a query function (e_{query}), which produces suggestions, and security state update functions that execute before (e_{acc}) and after (e_{res}) the monitored method. Each suggestion (irrs, oks, inss, repls, exns, and halts) also has its own constructor. For instance, the repls constructor takes a result object as an argument and the inss suggestion takes an action to execute as an argument. Each suggestion will be given a unique interpretation in the operational semantics.

5.2 Static Semantics

Figures 5.2, 5.3, and 5.4 present the rules for the language's static semantics. The main judgment, which types expressions, has the form $S; C \vdash e : \tau$ where S maps reference locations to their types and C maps variables to types. More precisely, S and C have the following forms.

$$\begin{array}{lrcl} \text{label stores} & S & ::= & \cdot \mid S, l : \tau \\ \text{variable contexts} & C & ::= & \cdot \mid C, x : \tau \mid C, f : \tau \end{array}$$

Whenever we add a new binding $x : \tau$ to the context, we implicitly alpha-vary x to ensure it does not clash with other variables in the context.

We have worked hard to make the static semantics a simple but faithful model of the implementation. In particular, notice in Figure 5.2 that well-typed policies contain a query method, which takes an action and returns a suggestion, and accept and result methods, which perform state updates. In addition, notice in Figure 5.2 that all actions share the same type (Act) regardless of the type of object they return

when invoked. Dynamically, the result of invoking an action is a value wrapped up as a result with type Res. Case analysis is used to safely extract the proper value. This choice allows policy objects to process and react to arbitrary actions. To determine the precise nature of any action and give it a more refined type, the policy will use pattern matching. We have a similar design for action results and replacement values.

The judgment for overall program states (shown in the middle of Figure 5.2) has the form $\vdash (\vec{F}, M, e_{\text{pol}}, e_{\text{app}}) : \tau$ where τ is the type of the application code e_{app}. This judgment relies on two additional judgments (shown in the bottom of Figure 5.2), which give types to a library of monitored functions $\vec{F}$ and types to locations in memory M.

Figure 5.3 presents the static semantics for case expressions and pattern matching. The auxiliary judgment $C \vdash p : (\tau; C')$ is used to check that a pattern p matches objects with type τ and binds variables with types given by C'.

Finally, Figure 5.4 displays the remaining, straightforward rules for the static semantics of standard expressions in our language.

$\boxed{S; C \vdash e : \tau}$

$$\frac{\begin{array}{c} S; C \vdash e_{\text{query}} : \text{Act} \to \text{Sug} \\ S; C \vdash e_{\text{acc}} : (\text{Act}, \text{Sug}) \to () \qquad S; C \vdash e_{\text{res}} : \text{Res} \to () \end{array}}{S; C \vdash \text{pol}(e_{\text{query}}, e_{\text{acc}}, e_{\text{res}}) : \text{Poly}}$$

$$\frac{}{S; C \vdash \text{irrs} : \text{Sug}} \qquad \frac{}{S; C \vdash \text{oks} : \text{Sug}}$$

$$\frac{S; C \vdash e : \text{Act}}{S; C \vdash \text{inss}(e) : \text{Sug}} \qquad \frac{S; C \vdash e : \text{Res}}{S; C \vdash \text{repls}(e) : \text{Sug}}$$

$$\frac{}{S; C \vdash \text{exns} : \text{Sug}} \qquad \frac{}{S; C \vdash \text{halts} : \text{Sug}}$$

$$\frac{C(f) = \tau_1 \to \tau_2 \quad S; C \vdash e : \tau_1}{S; C \vdash \text{act}(f, e) : \text{Act}} \qquad \frac{S; C \vdash e : \text{Act}}{S; C \vdash \text{invk } e : \text{Res}}$$

$$\frac{S; C \vdash e : \tau}{S; C \vdash \text{result}(e{:}\tau) : \text{Res}}$$

$\boxed{\vdash (\vec{F}, M, e_{\text{pol}}, e_{\text{app}}) : \tau}$

$$\frac{\begin{array}{c} \vdash \vec{F} : C \qquad C \vdash M : S \\ S; C \vdash e_{\text{pol}} : \text{Poly} \qquad S; C \vdash e_{\text{app}} : \tau \end{array}}{\vdash (\vec{F}, M, e_{\text{pol}}, e_{\text{app}}) : \tau}$$

$\boxed{\vdash \vec{F} : C}$

$$\frac{\begin{array}{c} \vec{F} = \text{fun} f_1(x_1{:}\tau_1){:}\tau_1'\{e_1\}, ..., \text{fun} f_n(x_n{:}\tau_n){:}\tau_n'\{e_n\} \\ C = f_1 : \tau_1 \to \tau_1', ..., f_n : \tau_n \to \tau_n' \\ \forall i \in \{1..n\} \;.\; \cdot\,; C, x_i : \tau_i \vdash e_i : \tau_i' \end{array}}{\vdash \vec{F} : C}$$

$\boxed{C \vdash M : S}$

$$\frac{\text{dom}(M) = \text{dom}(S) \quad \forall l \in \text{dom}(M) \;.\; S; C \vdash M(l) : S(l)}{C \vdash M : S}$$

Figure 5.2: Static semantics (rules for policies, suggestions, actions, and programs)

$\boxed{S; C \vdash e : \tau}$

$$\frac{\begin{array}{cc} S; C \vdash e_1 : \tau' & C \vdash p : (\tau'; C') \\ S; C, C' \vdash e_2 : \tau & S; C \vdash e_3 : \tau \end{array}}{S; C \vdash \text{case } e_1 \text{ of } (p \Rightarrow e_2 \mid \_ \Rightarrow e_3) : \tau}$$

$\boxed{C \vdash p : (\tau'; C')}$

$$\frac{}{C \vdash \text{pol}(x_1, x_2, x_3) : (\text{Poly};\ x_1 : \text{Act} \to \text{Sug}, x_2 : (\text{Act}, \text{Sug}) \to (), x_3 : \text{Res} \to ())}$$

$$\frac{}{C \vdash \text{oks} : (\text{Sug}; \cdot)} \qquad \frac{}{C \vdash \text{halts} : (\text{Sug}; \cdot)}$$

$$\frac{}{C \vdash \text{irrs} : (\text{Sug}; \cdot)} \qquad \frac{}{C \vdash \text{exns} : (\text{Sug}; \cdot)}$$

$$\frac{C \vdash p : (\text{Res}; C')}{C \vdash \text{repls}(p) : (\text{Sug}; C')} \qquad \frac{C \vdash p : (\text{Act}; C')}{C \vdash \text{inss}(p) : (\text{Sug}; C')}$$

$$\frac{C \vdash p : (\tau; C')}{C \vdash \text{result}(p : \tau) : (\text{Res}; C')} \qquad \frac{C(f) = \tau_1 \to \tau_2 \quad C \vdash p : (\tau_1; C')}{C \vdash \text{act}(f, p) : (\text{Act}; C')}$$

$$\frac{}{C \vdash \text{true} : (\text{Bool}; \cdot)} \qquad \frac{}{C \vdash \text{false} : (\text{Bool}; \cdot)}$$

$$\frac{\forall i \in \{1..n\} \;.\; C \vdash p_i : (\tau_i; C_i)}{C \vdash (p_1, ..., p_n) : ((\tau_1, ..., \tau_n); C_1, ..., C_n)} \qquad \frac{}{C \vdash x : (\tau; x : \tau)}$$

Figure 5.3: Static semantics (rules for case expressions)

$\boxed{S; C \vdash e : \tau}$

$$\frac{C(x) = \tau}{S; C \vdash x : \tau}$$

$$\frac{}{S; C \vdash \text{true} : \text{Bool}} \qquad \frac{}{S; C \vdash \text{false} : \text{Bool}}$$

$$\frac{\forall i \in \{1..n\} \;.\; S; C \vdash e_i : \tau_i}{S; C \vdash (e_1, .., e_n) : (\tau_1, ..., \tau_n)} \qquad \frac{S; C \vdash e_1 : () \quad S; C \vdash e_2 : \tau}{S; C \vdash e_1; e_2 : \tau}$$

$$\frac{S(l) = \tau}{S; C \vdash l : \tau \text{ Ref}} \qquad \frac{S; C \vdash e : \tau}{S; C \vdash \text{ref } e : \tau \text{ Ref}}$$

$$\frac{S; C \vdash e : \tau \text{ Ref}}{S; C \vdash !e : \tau} \qquad \frac{S; C \vdash e_1 : \tau \text{ Ref} \quad S; C \vdash e_2 : \tau}{S; C \vdash e_1 := e_2 : ()}$$

$$\frac{S; C, x : \tau \vdash e : \tau'}{S; C \vdash \lambda x{:}\tau.e \;:\; \tau \to \tau'} \qquad \frac{S; C \vdash e_1 : \tau_1 \to \tau_2 \quad S; C \vdash e_2 : \tau_1}{S; C \vdash e_1 \; e_2 : \tau_2}$$

$$\frac{}{S; C \vdash \text{abort} : \tau} \qquad \frac{}{S; C \vdash \text{raise exn} : \tau}$$

$$\frac{S; C \vdash e_1 : \tau \quad S; C \vdash e_2 : \tau}{S; C \vdash \text{try } e_1 \text{ with } e_2 : \tau}$$

Figure 5.4: Static semantics (standard rules)

5.3 Dynamic Semantics

To explain execution of monitored programs, we use a context-based semantics. The first step is to define a set of evaluation contexts E, which mark where a beta-reduction can occur. Our contexts specify a left-to-right, call-by-value evaluation order, as shown in Figure 5.5.

We specify execution through a pair of judgments, one for top-level evaluation (shown in the top of Figure 5.6) and one for basic reductions (shown in the bottom of Figure 5.6 and in Figures 5.7 and 5.8). The top-level judgment reveals that

evaluation contexts :

$$E ::= [\,] \mid (\vec{v}, E, \vec{e}) \mid E; e_2 \mid \text{ref } E \mid !E \mid E{:=}e_2 \mid v{:=}E \mid E\ e \mid v\ E \mid \text{pol}(E, e_2, e_3) \mid \text{pol}(v_1, E, e_3) \mid \text{pol}(v_1, v_2, E) \mid \text{inss}(E) \mid \text{repls}(E) \mid \text{act}(f, E) \mid \text{invk } E \mid \text{result}(E : \tau) \mid \text{case } E \text{ of } (p \Rightarrow e_2 \mid \_ \Rightarrow e_3) \mid \text{try } E \text{ with } e$$

Figure 5.5: Evaluation contexts

the policy expression is first reduced to a value before execution of the untrusted application code begins. Execution of many of the constructs is relatively straightforward. One exception is execution of function application, the rules for which are given in the bottom half of Figure 5.6. For ordinary functions, we use the usual capture-avoiding substitution. Monitored functions, on the other hand, may only be executed if they are wrapped up as actions and then invoked using the invk command. The invk command applies the query method to discover the suggestion the current policy makes and then interprets the suggestion. Notice, for instance, that to respond to the irrelevant suggestion (irrs), the application simply proceeds to execute the body of the security-relevant action. To respond to the OK suggestion (oks), the application first calls the policy's accept method, then executes the security-relevant action before calling the policy's result method, and finally returns the result of executing the security-relevant action.

The other beta reductions (i.e., besides the ones for function application) are straightforward and appear in Figures 5.7 and 5.8. The rules for case expressions and pattern matching (Figure 5.7) rely on an auxiliary judgment $v \sim p : V$, which holds when value v matches pattern p, and the pattern matching produces capture-avoiding variable substitutions V.

$\boxed{(\vec{F}, M, e_{\text{pol}}, e_{\text{app}}) \mapsto (\vec{F}, M', e'_{\text{pol}}, e'_{\text{app}})}$

$$\frac{(\vec{F}, M, \text{Triv}, e) \rightarrow_\beta (M', e')}{(\vec{F}, M, E[e], e_{\text{app}}) \mapsto (\vec{F}, M', E[e'], e_{\text{app}})}$$

where $\text{Triv} = \text{pol}(\lambda x{:}\text{Act.irrs}, \lambda x{:}(\text{Act}, \text{Sug}).(), \lambda x{:}\text{Res.}())$

$$\frac{(\vec{F}, M, v_{\text{pol}}, e) \rightarrow_\beta (M', e')}{(\vec{F}, M, v_{\text{pol}}, E[e]) \mapsto (\vec{F}, M', v_{\text{pol}}, E[e'])}$$

$\boxed{(\vec{F}, M, v_{\text{pol}}, e) \rightarrow_\beta (M', e')}$

$$\frac{}{(\vec{F}, M, v_{\text{pol}}, (\lambda x{:}\tau.e)v) \rightarrow_\beta (M, e[v/x])}$$

$$\frac{F_i \in \vec{F} \quad F_i = \text{fun}f(x{:}\tau_1){:}\tau_2\{e\}}{(\vec{F}, M, v_{\text{pol}}, \text{invk act}(f, v)) \rightarrow_\beta (M, \text{Wrap}(v_{\text{pol}}, F_i, v))}$$

where $\text{Wrap}(\text{pol}(v_{\text{query}}, v_{\text{acc}}, v_{\text{res}}), \text{fun}f(x{:}\tau_1){:}\tau_2\{e\}, v) =$

```
  let s = v_query(act(f, v)) in
  case s of
     irrs      ⇒  let x = v in result(e:τ2)
   | oks       ⇒  v_acc(act(f, v), s);
                  let x = v in let r = result(e:τ2) in v_res r;  r
   | repls(r)  ⇒  v_acc(act(f, v), s);  r
   | exns      ⇒  v_acc(act(f, v), s);  raise exn
   | inss(a)   ⇒  v_acc(act(f, v), s);  v_res(invk a);  invk act(f, v)
   | _         ⇒  abort
```

Figure 5.6: Dynamic semantics (policy and target steps; beta steps for functions)

$\boxed{(\vec{F}, M, v_{\text{pol}}, e) \rightarrow_\beta (M', e')}$

$$\frac{v \sim p : V}{(\vec{F}, M, v_{\text{pol}}, \text{case } v \text{ of } (p \Rightarrow e_2 \mid \_ \Rightarrow e_3)) \rightarrow_\beta (M, e_2[V])}$$

$$\frac{\neg \exists V.\ v \sim p : V}{(\vec{F}, M, v_{\text{pol}}, \text{case } v \text{ of } (p \Rightarrow e_2 \mid \_ \Rightarrow e_3)) \rightarrow_\beta (M, e_3)}$$

$\boxed{v \sim p : V}$

$$\frac{}{\text{pol}(v_1, v_2, v_3) \sim \text{pol}(x_1, x_2, x_3) : v_1/x_1, v_2/x_2, v_3/x_3}$$

$$\frac{}{\text{oks} \sim \text{oks} : \cdot} \qquad \frac{}{\text{halts} \sim \text{halts} : \cdot}$$

$$\frac{}{\text{irrs} \sim \text{irrs} : \cdot} \qquad \frac{}{\text{exns} \sim \text{exns} : \cdot}$$

$$\frac{v \sim p : V}{\text{repls}(v) \sim \text{repls}(p) : V} \qquad \frac{v \sim p : V}{\text{inss}(v) \sim \text{inss}(p) : V}$$

$$\frac{v \sim p : V}{\text{result}(v : \tau) \sim \text{result}(p : \tau) : V} \qquad \frac{v \sim p : V}{\text{act}(f, v) \sim \text{act}(f, p) : V}$$

$$\frac{}{\text{true} \sim \text{true} : \cdot} \qquad \frac{}{\text{false} \sim \text{false} : \cdot}$$

$$\frac{}{v \sim x : v/x} \qquad \frac{\forall i \in \{1..n\}\ .\ v_i \sim p_i : V_i}{(v_1, ..., v_n) \sim (p_1, ..., p_n) : V_1, ..., V_n}$$

Figure 5.7: Dynamic semantics (beta steps for case expressions)

$$\boxed{(\vec{F}, M, v_{\text{pol}}, e) \rightarrow_\beta (M', e')}$$

$$\frac{l \notin \text{dom}(M)}{(\vec{F}, M, v_{\text{pol}}, \text{ref } v) \rightarrow_\beta ((M, l : v), l)} \qquad \frac{M(l) = v}{(\vec{F}, M, v_{\text{pol}}, !l) \rightarrow_\beta (M, v)}$$

$$\frac{}{(\vec{F}, M, v_{\text{pol}}, (); e) \rightarrow_\beta (M, e)} \qquad \frac{}{(\vec{F}, M, v_{\text{pol}}, l{:=}v) \rightarrow_\beta ([l \rightarrow v]M, ())}$$

$$\frac{}{(\vec{F}, M, v_{\text{pol}}, \text{try } v \text{ with } e) \rightarrow_\beta (M, v)}$$

$$\frac{E \neq E'[\text{try } E'' \text{ with } e']}{(\vec{F}, M, v_{\text{pol}}, \text{try } E[\text{raise exn}] \text{ with } e) \rightarrow_\beta (M, e)}$$

Figure 5.8: Dynamic semantics (standard beta steps)

5.4 Semantics-based Observations

The formal semantics gives insight into some of the subtler elements of our implementation, which are important both to system users and to us as implementers.

For example, one might want to consider what happens if a program monitor raises but does not catch an exception (such as a null pointer exception). Tracing through the operational semantics, one can see that the exception will percolate from the monitor into the application itself. If this behavior is undesired, a security programmer can create a top-level superpolicy that catches all exceptions raised by the other policies and deals with them as the programmer sees fit.

As another example, analysis of the operational semantics shows a corner case in which we are unable to fully obey the principle of complete mediation. During the first stage of execution, while the policy itself is evaluated, monitored functions are only protected by a trivial policy that accepts all actions because the actual policy we want to enforce is the one being initialized. Policy writers need to be aware of this unavoidable behavior in order to implement policies correctly.

5.5 Type Safety

To check that our language is sound, we have proven a standard type-safety result in terms of Preservation and Progress theorems. Type safety is an important result because it implies that statically well-typed programs do not "get stuck" operationally.

We state below the lemmas and theorems in our proof of type safety, and we provide the proof technique for each. For the lemmas and theorems with non-trivial proofs, we also provide proofs for a selection of interesting cases.

Lemma 11 (Variable Substitution)

If $S; C, x : \tau' \vdash e : \tau$ *and* $S; C \vdash e' : \tau'$ *then* $S; C \vdash e[e'/x] : \tau$.

Proof By induction on the derivation of $S; C, x : \tau' \vdash e : \tau$. ■

Lemma 12 (Store Substitution)

If $C \vdash M : S$ *and* $S(l) = \tau$ *and* $S; C \vdash v : \tau$ *then* $C \vdash [l \to v]M : S$.

Proof Immediate by the sole typing rule for $C \vdash M : S$. ■

Lemma 13 (Weakening)

If $S; C \vdash e : \tau$ *and* S' *extends* S *and* C' *extends* C *then* $S'; C' \vdash e : \tau$.

Proof By induction on the derivation of $S; C \vdash e : \tau$. ■

Lemma 14 (Inversion of Typing)

Every typing rule is invertible. That is, if the conclusion of any typing rule (in Figures 5.2, 5.3, and 5.4) holds then its premises must also hold. For example, if

$S; C \vdash \text{inss}(e) : \text{Sug}$ *then* $S; C \vdash e : \text{Act}$*; as another example, if* $C \vdash \text{act}(f, p) : (\text{Act}; C')$ *then* $C(f) = \tau_1 \to \tau_2$ *and* $C \vdash p : (\tau_1; C')$*.*

Proof Immediate by inspection of the typing rules. ■

Lemma 15 (Canonical Forms)

If $S; C \vdash v : \tau$ *then*

- $\tau = \text{Bool}$ *implies* $v = \text{true}$ *or* $v = \text{false}$
- $\tau = (\tau_1, ..., \tau_n)$ *implies* $v = (v_1, ..., v_n)$
- $\tau = \tau'$ Ref *implies* $v = l$
- $\tau = \tau_1 \to \tau_2$ *implies* $v = \lambda x{:}\tau_1.e$
- $\tau = \text{Poly}$ *implies* $v = \text{pol}(v_{\text{query}}, v_{\text{acc}}, v_{\text{res}})$
- $\tau = \text{Sug}$ *implies* $v = \text{irrs}$ *or* $v = \text{oks}$ *or* $v = \text{inss}(\text{act}(f, v))$ *or* $v = \text{exns}$ *or* $v = \text{halts}$ *or* $v = \text{repls}(\text{result}(v{:}\tau))$
- $\tau = \text{Act}$ *implies* $v = \text{act}(f, v)$
- $\tau = \text{Res}$ *implies* $v = \text{result}(v{:}\tau)$

Proof By induction on the derivation of $S; C \vdash v : \tau$, using the definition of values (given in Figure 5.1). ■

Definition 16 (Well-typed Context)

A context E *is well typed, written* $S; C \vdash E_\tau : \tau'$*, if and only if* $S; C, x : \tau \vdash E[x] : \tau'$ *(where* x *is not a free variable in* E*).*

Lemma 17 (Well-typed, Filled Context)

If $S; C \vdash E_\tau : \tau'$ *and* $S; C \vdash e : \tau$ *then* $S; C \vdash E[e] : \tau'$.

Proof Immediate by Definition 16 (well-typed context) and Lemma 11 (variable substitution). ■

Lemma 18 (Context Decomposition)

If $S; C \vdash E[e] : \tau$ *then there exists a* τ' *such that* $S; C \vdash E_{\tau'} : \tau$ *and* $S; C \vdash e : \tau'$.

Proof By induction on the structure of E. We show the case for $E = \text{invk } E'$. The other cases are similar (except the case $E = [\,]$, which is trivial).

Case $E = \text{invk } E'$:

By assumption,

$$E = \text{invk } E' \tag{1}$$

$$S; C \vdash E[e] : \tau \tag{2}$$

By (1) and (2),

$$S; C \vdash \text{invk } E'[e] : \tau \tag{3}$$

By (3) and Lemma 14 (inversion of the typing rule for invk),

$$S; C \vdash E'[e] : \text{Act} \tag{4}$$

By the inductive hypothesis and (4),

$$S; C \vdash E'_{\tau''} : \text{Act} \tag{5}$$

$$S; C \vdash e : \tau'' \tag{6}$$

By (5) and Definition 16 (well-typed context),

$$S; C, x : \tau'' \vdash E'[x] : \text{Act} \qquad (x \notin FV(E')) \tag{7}$$

By (7) and the typing rule for invk,

$$S; C, x : \tau'' \vdash \text{invk } E'[x] : \text{Res} \tag{8}$$

By (1) and (8),

$$S; C, x : \tau'' \vdash E[x] : \text{Res} \tag{9}$$

By (9) and Definition 16 (well-typed context),

$$S; C \vdash E_{\tau''} : \text{Res} \tag{10}$$

By (3) and the typing rule for invk,

$$\tau = \text{Res} \tag{11}$$

Result is from (6), (10), and (11). ■

Definition 19 (Well-typed Substitutions)

A sequence of variable substitutions V has type C', written $S; C \vdash V : C'$, if and only if for all $x \in \text{dom}(C')$ there exists v such that $v/x \in V$ and $S; C \vdash v : C'(x)$.

Lemma 20 (Pattern Types)

If $S; C \vdash v : \tau'$ and $v \sim p : V$ and $C \vdash p : (\tau'; C')$ then $S; C \vdash V : C'$.

Proof By induction on $v \sim p : V$. We show two cases; the others are similar.

Case $p = \text{pol}(x_1, x_2, x_3)$:

By assumption,

$$S; C \vdash v : \tau' \tag{1}$$

$$v = \text{pol}(v_1, v_2, v_3) \tag{2}$$

$$V = v_1/x_1, v_2/x_2, v_3/x_3 \tag{3}$$

$$C' = x_1 : \text{Act} \to \text{Sug}, x_2 : (\text{Act}, \text{Sug}) \to (), x_3 : \text{Res} \to () \tag{4}$$

By (1), (2), and Lemma 14 (inversion of the typing rule for policies),

$$S; C \vdash v_1 : \text{Act} \to \text{Sug} \tag{5}$$

$$S; C \vdash v_2 : (\text{Act}, \text{Sug}) \to () \tag{6}$$

$$S; C \vdash v_3 : \text{Res} \to () \tag{7}$$

By (3), (4), (5), (6), and (7),

$$\forall x \in \mathrm{dom}(C')\ \exists v\ .\ (v/x\ \in V\ \wedge\ S; C \vdash v : C'(x)) \tag{8}$$

By (8) and Definition 19 (well-typed substitutions),

$$S; C \vdash V : C' \tag{9}$$

Case $p = \mathrm{act}(f, p')$:

By assumption,

$$S; C \vdash v : \tau' \tag{1}$$

$$v = \mathrm{act}(f, v') \tag{2}$$

$$v' \sim p' : V \tag{3}$$

$$C \vdash p : (\mathrm{Act}; C') \tag{4}$$

By Lemma 14 (inversion of the typing rule for (4)),

$$C \vdash p' : (\tau_1; C') \quad (\text{where } C(f) = \tau_1 \to \tau_2) \tag{5}$$

By (1), (2), and Lemma 14 (inversion of the typing rule for $\mathrm{act}(f, v')$),

$$S; C \vdash v' : \tau_1 \tag{6}$$

By (3), (5), (6), and the inductive hypothesis,

$$S; C \vdash V : C' \tag{7}$$

■

Lemma 21 (Multiple Substitutions)

If $S; C \vdash V : C'$ and $S; C, C' \vdash e : \tau$ then $S; C \vdash e[V] : \tau$.

Proof By induction on the length of V, using Lemma 11 (variable substitution).

■

Lemma 22 (Basic Decomposition)

If $S; C \vdash e : \tau$ and $\vdash \vec{F} : C$ and $C \vdash M : S$ and $v_{\mathrm{pol}} = \mathrm{pol}(v_{\mathrm{query}}, v_{\mathrm{acc}}, v_{\mathrm{res}})$ then either:

- *e is a value v, or*
- *e can be decomposed into $E[e']$ such that one of the following is true.*
  - $(\vec{F}, M, v_{\text{pol}}, e') \rightarrow_\beta (M', e'')$, *for some* M' *and* e''
  - $e' = \text{raise exn}$ *and* $E \neq E'[\text{try } E'' \text{ with } e'']$
  - $e' = \text{abort}$

Proof By induction on the derivation of $S; C \vdash e : \tau$. ■

Lemma 23 (Policy Decomposition)

If $\vdash (\vec{F}, M, e_{\text{pol}}, e_{\text{app}}) : \tau$ *then either:*

- e_{pol} *is a value* v_{pol}, *or*
- e_{pol} *can be decomposed into* $E[e]$ *such that one of the following is true.*
  - $(\vec{F}, M, \text{Triv}, e) \rightarrow_\beta (M', e')$, *for some* M' *and* e', *where* Triv *is the trivial policy defined in Figure 5.6*
  - $e = \text{raise exn}$ *and* $E \neq E'[\text{try } E'' \text{ with } e'']$
  - $e = \text{abort}$

Proof Immediate by Lemma 14 (inversion of $\vdash (\vec{F}, M, e_{\text{pol}}, e_{\text{app}}) : \tau$) and Lemma 22 (basic decomposition). ■

Lemma 24 (Application Decomposition)

If $\vdash (\vec{F}, M, v_{\text{pol}}, e_{\text{app}}) : \tau$ *then either:*

- e_{app} *is a value* v_{app}, *or*

- e_{app} *can be decomposed into* $E[e]$ *such that one of the following is true.*
  - $(\vec{F}, M, v_{\text{pol}}, e) \rightarrow_\beta (M', e')$, *for some* M' *and* e'
  - $e = \text{raise exn}$ *and* $E \neq E'[\text{try } E'' \text{ with } e'']$
  - $e = \text{abort}$

Proof Immediate by Lemma 14 (inversion of $\vdash (\vec{F}, M, v_{\text{pol}}, e_{\text{app}}) : \tau$), Lemma 15 (canonical forms for $S; C \vdash v_{\text{pol}} : \text{Poly}$), and Lemma 22 (basic decomposition). ■

Lemma 25 (β Preservation)
If $\vdash (\vec{F}, M, v_{\text{pol}}, e_{\text{app}}) : \tau$ *and* $\vdash \vec{F} : C$ *and* $C \vdash M : S$ *and* $(\vec{F}, M, v_{\text{pol}}, e_{\text{app}}) \rightarrow_\beta (M', e'_{\text{app}})$ *then there exists an* S' *extending* S *such that* $S'; C \vdash e'_{\text{app}} : \tau$ *and* $C \vdash M' : S'$.

Proof By induction on $(\vec{F}, M, v_{\text{pol}}, e_{\text{app}}) \rightarrow_\beta (M', e'_{\text{app}})$. We show two non-trivial cases; the others are similar.

Case $e_{\text{app}} = \text{invk act}(f, v)$:

By assumption,

$$\vdash (\vec{F}, M, v_{\text{pol}}, e_{\text{app}}) : \tau \tag{1}$$
$$\vdash \vec{F} : C \tag{2}$$
$$C \vdash M : S \tag{3}$$
$$(\vec{F}, M, v_{\text{pol}}, e_{\text{app}}) \rightarrow_\beta (M', e'_{\text{app}}) \tag{4}$$
$$e_{\text{app}} = \text{invk act}(f, v) \tag{5}$$
$$\text{fun} f(x{:}\tau_1){:}\tau_2\{e\} \in \vec{F} \tag{6}$$
$$e'_{\text{app}} = \text{Wrap}(v_{\text{pol}}, \text{fun} f(x{:}\tau_1){:}\tau_2\{e\}, v) \tag{7}$$

By (2), (3), and Lemma 14 (inversion of the typing rule for (1)),

$$S; C \vdash v_{\text{pol}} : \text{Poly} \tag{8}$$

$$S; C \vdash e_{\text{app}} : \tau \tag{9}$$

By (5), (9), and the typing rule for invk,

$$S; C \vdash e_{\text{app}} : \text{Res} \tag{10}$$

By (5), (10), and Lemma 14 (inversion of the typing rule for invk),

$$S; C \vdash \text{act}(f, v) : \text{Act} \tag{11}$$

By (11) and Lemma 14 (inversion of the typing rule for $\text{act}(f, v)$),

$$C(f) = \tau_1' \to \tau_2' \tag{12}$$

$$S; C \vdash v : \tau_1' \tag{13}$$

By (6), (12), and Lemma 14 (inversion of the typing rule for (2)),

$$\tau_1' = \tau_1 \tag{14}$$

$$\tau_2' = \tau_2 \tag{15}$$

$$\cdot \; ; \; C, x : \tau_1 \vdash e : \tau_2 \tag{16}$$

By (12), (13), (14), and (15),

$$C(f) = \tau_1 \to \tau_2 \tag{17}$$

$$S; C \vdash v : \tau_1 \tag{18}$$

By (8) and Lemma 15 (canonical forms),

$$v_{\text{pol}} = \text{pol}(v_{\text{query}}, v_{\text{acc}}, v_{\text{res}}) \tag{19}$$

By (8), (19), and Lemma 14 (inversion of the typing rule for Poly),

$$S; C \vdash v_{\text{query}} : \text{Act} \to \text{Sug} \tag{20}$$

$$S; C \vdash v_{\text{acc}} : (\text{Act}, \text{Sug}) \to () \tag{21}$$

$$S; C \vdash v_{\text{res}} : \text{Res} \to () \tag{22}$$

By (7), (16), (17), (18), (19), (20), (21), (22), and the definition of Wrap,

$$S; C \vdash e_{\text{app}}' : \text{Res} \tag{23}$$

By (9) and (10),

$$\tau = \text{Res} \tag{24}$$

By (3), (23), and (24),

$$S; C \vdash e'_{\text{app}} : \tau \ \wedge \ C \vdash M : S \tag{25}$$

Case $e_{\text{app}} = \text{case } v \text{ of } (p \Rightarrow e_2 \mid \_ \Rightarrow e_3)$ and $v \sim p : V$:

By assumption,

$$\vdash (\vec{F}, M, v_{\text{pol}}, e_{\text{app}}) : \tau \tag{1}$$

$$\vdash \vec{F} : C \tag{2}$$

$$C \vdash M : S \tag{3}$$

$$(\vec{F}, M, v_{\text{pol}}, e_{\text{app}}) \rightarrow_\beta (M, e_2[V]) \tag{4}$$

$$v \sim p : V \tag{5}$$

$$e_{\text{app}} = \text{case } v \text{ of } (p \Rightarrow e_2 \mid \_ \Rightarrow e_3) \tag{6}$$

By (2), (3), and Lemma 14 (inversion of the typing rule for (1)),

$$S; C \vdash e_{\text{app}} : \tau \tag{7}$$

By (6), (7), and Lemma 14 (inversion of the typing rule for case expressions),

$$S; C \vdash v : \tau' \tag{8}$$

$$C \vdash p : (\tau'; C') \tag{9}$$

$$S; C, C' \vdash e_2 : \tau \tag{10}$$

By (5), (8), (9), and Lemma 20 (pattern types),

$$S; C \vdash V : C' \tag{11}$$

By (10), (11), and Lemma 21 (multiple substitutions),

$$S; C \vdash e_2[V] : \tau \tag{12}$$

Result is from (3), (4), and (12). ■

Theorem 26 (Preservation)

If $\vdash (\vec{F}, M, e_{\text{pol}}, e_{\text{app}}) : \tau$ *and* $(\vec{F}, M, e_{\text{pol}}, e_{\text{app}}) \mapsto (\vec{F}, M', e'_{\text{pol}}, e'_{\text{app}})$ *then* $\vdash (\vec{F}, M', e'_{\text{pol}}, e'_{\text{app}}) : \tau$.

Proof Examination of the rules for $(\vec{F}, M, e_{\text{pol}}, e_{\text{app}}) \mapsto (\vec{F}, M', e'_{\text{pol}}, e'_{\text{app}})$ demonstrates that either $e_{\text{pol}} = E[e]$ or $e_{\text{pol}} = v_{\text{pol}}$ (that is, either the policy or the application is being evaluated). The two cases are similar; we show the first case.

Case $e_{\text{pol}} = E[e]$:

By assumption,

$$\vdash (\vec{F}, M, e_{\text{pol}}, e_{\text{app}}) : \tau \tag{1}$$

$$e_{\text{pol}} = E[e] \tag{2}$$

$$(\vec{F}, M, \text{Triv}, e) \rightarrow_{\beta} (M', e') \tag{3}$$

$$e'_{\text{pol}} = E[e'] \tag{4}$$

$$e_{\text{app}} = e'_{\text{app}} \tag{5}$$

By (2) and Lemma 14 (inversion of the typing rule for (1)),

$$\vdash \vec{F} : C \tag{6}$$

$$C \vdash M : S \tag{7}$$

$$S; C \vdash E[e] : \text{Poly} \tag{8}$$

$$S; C \vdash e_{\text{app}} : \tau \tag{9}$$

By (8) and Lemma 18 (context decomposition),

$$S; C \vdash E_{\tau'} : \text{Poly} \tag{10}$$

$$S; C \vdash e : \tau' \tag{11}$$

By the definition of Triv,

$$\cdot \; ; \; \cdot \; \vdash \text{Triv} : \text{Poly} \tag{12}$$

By (5) and (9),

$$S; C \vdash e'_{\text{app}} : \tau \tag{13}$$

By (6), (7), (11), (12), and the typing rule for program configurations,

$$\vdash (\vec{F}, M, \text{Triv}, e) : \tau' \tag{14}$$

By (3), (6), (7), (14), and Lemma 25 (β preservation),

$$S' \text{ extends } S \tag{15}$$

$$S'; C \vdash e' : \tau' \tag{16}$$

$$C \vdash M' : S' \tag{17}$$

By (10), (15), Definition 16 (well-typed context), and Lemma 13 (weakening),

$$S'; C \vdash E_{\tau'} : \text{Poly} \tag{18}$$

By (4), (16), (18), and Lemma 17 (well-typed, filled context),

$$S'; C \vdash e'_{\text{pol}} : \text{Poly} \tag{19}$$

By (13) and Lemma 13 (weakening),

$$S'; C \vdash e'_{\text{app}} : \tau \tag{20}$$

By (6), (17), (19), (20), and the typing rule for program configurations,

$$\vdash (\vec{F}, M', e'_{\text{pol}}, e'_{\text{app}}) : \tau \tag{21}$$

■

Definition 27 (Finished Programs)

A program configuration $(\vec{F}, M, e_{\text{pol}}, e_{\text{app}})$ *is "finished" if and only if at least one of the following is true.*

- e_{pol} *and* e_{app} *are values*
- $e_{\text{pol}} = E[\text{abort}]$ *or* $e_{\text{app}} = E[\text{abort}]$
- $e_{\text{pol}} = E[\text{raise exn}]$ *or* $e_{\text{app}} = E[\text{raise exn}]$, *where* $E \neq E'[\text{try } E'' \text{ with } e]$

Theorem 28 (Progress)

If $\vdash (\vec{F}, M, e_{\text{pol}}, e_{\text{app}}) : \tau$ *then either* $(\vec{F}, M, e_{\text{pol}}, e_{\text{app}})$ *is* finished *or there exists a program configuration* $(\vec{F}, M', e'_{\text{pol}}, e'_{\text{app}})$ *such that* $(\vec{F}, M, e_{\text{pol}}, e_{\text{app}}) \mapsto (\vec{F}, M', e'_{\text{pol}}, e'_{\text{app}})$.

Proof By applying Lemma 23 (policy decomposition) to the assumption that $\vdash (\vec{F}, M, e_{\text{pol}}, e_{\text{app}}) : \tau$, we have either $e_{\text{pol}} = v_{\text{pol}}$ or $e_{\text{pol}} = E[e]$ such that $e =$ raise exn (where $E \neq E'[\text{try } E'' \text{ with } e'']$) or $e =$ abort or $(\vec{F}, M, \text{Triv}, e) \rightarrow_\beta (M', e')$. When $e_{\text{pol}} = E[e]$ such that $e =$ raise exn or $e =$ abort, the program $(\vec{F}, M, e_{\text{pol}}, e_{\text{app}})$ is finished. When $e_{\text{pol}} = E[e]$ and $(\vec{F}, M, \text{Triv}, e) \rightarrow_\beta (M', e')$, we have $(\vec{F}, M, e_{\text{pol}}, e_{\text{app}}) \mapsto (\vec{F}, M', E[e'], e_{\text{app}})$, as required.

When $e_{\text{pol}} = v_{\text{pol}}$, Lemma 24 (application decomposition) implies that either $e_{\text{app}} = v_{\text{app}}$ or $e_{\text{app}} = E[e]$ such that $e =$ raise exn (where $E \neq E'[\text{try } E'' \text{ with } e'']$) or $e =$ abort or $(\vec{F}, M, v_{\text{pol}}, e) \rightarrow_\beta (M', e')$. All of these possibilities correspond to finished program configurations, except the case where $e_{\text{pol}} = v_{\text{pol}}$ and $e_{\text{app}} = E[e]$ and $(\vec{F}, M, v_{\text{pol}}, e) \rightarrow_\beta (M', e')$. In this case, we have $(\vec{F}, M, e_{\text{pol}}, e_{\text{app}}) \mapsto (\vec{F}, M', e_{\text{pol}}, E[e'])$, as required. ■

Chapter 6

Conclusions

This thesis has taken steps to improve our understanding both of the space of policies program monitors can enforce and of how to design a practical language and system for specifying and enforcing monitors' policies. Our final chapter summarizes our primary contributions (Section 6.1), enumerates several directions for future work (Section 6.2), and makes closing remarks (Section 6.3).

6.1 Summary

As outlined in Section 1.2, we have made four principal contributions.

1. A Formal Framework for Reasoning About Enforcement Starting from standard definitions of policies and properties, we have created formal models of sophisticated program-monitoring mechanisms (edit automata) and defined how they enforce properties by transforming execution sequences. We also introduced notation, refined by experience, for conveniently yet precisely specifying monitors, properties, systems, and executions.

Building this formal framework is a primary contribution of our work. Such a foundation enables us to be clear and explicit regarding our basic assumptions about what constitutes a policy, a monitor, and enforcement of a policy by a monitor. More importantly, without a formal foundation we could not demonstrate rigorously that particular mechanisms enforce particular policies.

2. Analysis of Policies Enforceable by Monitors When considering the space of security properties enforceable by monitoring potentially nonterminating programs, we have found that a simple variety of monitor enforces exactly the set of computable and satisfiable safety properties while a more powerful variety can enforce any computable infinite renewal property that is satisfied by the empty sequence. Because the set of renewal properties is a strict superset of the safety properties, there exist program monitors that can enforce non-safety properties. We have shown how, when given any reasonable renewal property, to construct an edit automaton that provably enforces that property.

Awareness of formally proven bounds on the power of security mechanisms facilitates our understanding of policies themselves and the mechanisms we need to enforce them. For example, observing that a stack-inspection policy is really just an access-control property (where access is granted or denied based on the history of function calls and returns), which in turn is clearly a safety property, makes it immediately obvious that simple monitors modeled by truncation automata are sufficient for enforcing stack-inspection policies. Similarly, if we can observe that infinite executions in a property specifying how users log in are valid if and only if they contain infinitely many valid prefixes, then we immediately know that monitors based on edit automata can enforce this renewal property. We hope that with con-

tinued research into the formal enforcement bounds of various security mechanisms, security architects will be able to pull from their enforcement "toolbox" exactly the right sorts of mechanisms needed to enforce the policies at hand.

3. A Language for Specifying Run-time Policies We have developed a programming methodology for writing complex security policies. The design is quite different from existing policy-specification languages in its division of policies into effectless methods that make suggestions regarding how to handle trigger actions and effectful methods that are called when the policy's suggestions are followed. This design facilitates composition and allows complex security policies to be specified more simply as compositions of smaller subpolicy modules. We have implemented our design in a language and system called Polymer and demonstrated its practicality by building a complex security policy for email clients from simple, modular, and reusable policies.

4. Formal Semantics for Our Policy-specification Language We have made unambiguous the meaning of the Polymer language by giving it a formal semantics. Because the semantics faithfully models our implementation, it provides insight into the detailed workings of our implemented system. Security engineers can consult the language's semantics in order to learn exactly how their Polymer policies operate in tandem with a target application and how to organize their policies to be well typed. We have proven that our language is sound by demonstrating a standard type-safety result in terms of preservation and progress lemmas. This type-safety result assures Polymer users that their well-typed programs will not "get stuck" operationally.

6.2 Future Work

There are many possibilities for extending our work to address open problems. We enumerate some of the possibilities.

Practical Constraints on Theoretical Monitors Sections 3.2.2 and 4.4.3 discuss a practical limitation of monitors absent from our current theoretical model: monitors often do not have the same computational capabilities as the machine that executes target actions. This limitation of real monitors implies that some actions cannot be suppressed (i.e., the monitor cannot "feign" an action), and some actions cannot be inserted (i.e., the monitor cannot obtain information needed to invoke an action). One easily imagined extension of our current framework is to incorporate sets of unsuppressible and uninsertable actions into system definitions and to analyze which properties edit automata can enforce under those conditions. This extension would make our model more precise, though significantly more complex.

Several additional practical constraints could be placed on monitors; that is, we could consider bounding the resources available to monitors even beyond making certain actions unsuppressible or uninsertable. For instance, Fong has shown that limiting the memory available to monitors induces limits on the properties they can enforce [23]. We might ask similar questions of time bounds on monitors: Are there useful properties that require super-polynomial monitoring time to enforce? How can we add real-time constraints to our model to reflect practical limits on the amount of real time monitors may consume, and how do these constraints affect the enforcement of real-time policies?

Formally Linking Edit Automata with Polymer Policies Section 4.4.3 gave an informal description of the ability of Polymer policies to implement edit automata: Polymer's `InsSug` *inserts* an action, and its `ReplSug`, `ExnSug`, and `HaltSug` are different ways to *suppress* actions (and `IrrSug` and `OKSug` *accept* actions). This implementation is simple and intuitive, but it would be nice to have a formally proven bisimulation between the operational semantics of edit automata and Polymer policies. Proving such a bisimulation would be interesting because it would tie our practical monitor specifications to properties enforceable by edit automata, allowing us to describe formally the space of policies enforceable in Polymer.

Transactional Policies One could view all renewal policies as transactional in nature, where the definition of whether a sequence of actions constitutes a valid transaction may depend upon the entire history of the current execution. The ATM-logging and file-contents policies of Section 4.4.3 implement transactions: we attempt to commit a sequence of actions atomically only when those actions, taken together, form a valid transaction (in the ATM or file system).

In addition, one could view the Polymer language as enabling policy composition via transactional policy updates. The separation of policies into effectless `query` and effectful `result` methods implements a form of rollback so that the highest-level superpolicy can commit to one suggestion atomically, without directly managing and rolling back subpolicy state and effects (which may be irrevocable).

Hence, strong ties seem to exist between run-time policy enforcement and transactions. In the future, it would be interesting to explore these ties further and to examine in exactly which ways languages with transactional support (e.g., [26]) further facilitate run-time policy specification and enforcement.

Concurrency We should consider adding concurrency to our models of edit automata; in such a context, executions might be partial orderings of actions rather than total orderings. Similarly, we have avoided considering the difficulties associated with concurrency in Polymer. We plan to make Polymer policies thread safe by adding a locking mechanism in the interpreter of the highest-level policy's suggestions (see Figure 4.1); the interpreter will obtain a lock before initiating a `query` and hold that lock until the corresponding `accept` method has returned.

Combinator Analysis As discussed in Section 4.3.3, we designed the Polymer language to permit arbitrary policy composition. This generality is useful because the definition of which combinators are the "right" ones to have available is user and application specific. For example, one set of combinators might be the minimal necessary to express all compositions of a common sort of policy (such as access-control policies), while different sets of combinators might be guaranteed to terminate or to satisfy other useful properties such as associativity and commutativity. Although Polymer permits general policy compositions, it would be interesting in the future to analyze particular sets of combinators and prove that they satisfy these sorts of properties. Krishnan has already made progress on formalizing many of our combinators [34].

Polymer GUI Polymer policies, while expressive, have to be written at too low of a level (at the level of Java source code) to be convenient for many users who might benefit from creating custom policy compositions. An interesting avenue for future work would be to extend Polymer with a graphical user interface (GUI) that would allow, for example, system administrators to easily form provably safe policy hierarchies using prepackaged base policies and policy combinators.

6.3 Closing Remarks

Mechanisms for enforcing software security typically incorporate some sort of program monitoring. Firewalls, virtual machines, operating systems, network scanners, intrusion detection systems, and antivirus and auditing tools clearly make extensive use of run-time monitoring to enforce security. Even "static" mechanisms, such as type-safe-language compilers and verifiers, often ensure that programs contain appropriate dynamic checks *inlined* into the code. The inlined checks implement monitors, guaranteeing for example that programs obey run-time memory-safety [29, 43] or control-flow [1, 2] policies.

Given their abundance and practicality as enforcement mechanisms, it seems strange that we understand relatively little of monitors' actual enforcement capabilities and have relatively primitive tools for designing, reasoning about, and implementing monitors. Even basic results, such as that practical monitors can sometimes enforce liveness properties (Section 4.4.3), surprise us.

By continuing to explore the capabilities and designs of various types of program monitors, we hope to improve our fundamental knowledge of these important mechanisms and make them easier to use and verify. As a long-term goal, we would like to see a wide variety of static and dynamic mechanisms, and the ways in which they can be composed to enforce policies, understood so deeply that tools and techniques will exist for generating, when possible, efficient mechanisms that provably enforce given policies.

Bibliography

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity: Principles, implementations, and applications. In *ACM Conference on Computer and Communications Security*, November 2005.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. A theory of secure control flow. In *International Conference on Formal Engineering Methods*, November 2005.

[3] Martín Abadi and Cédric Fournet. Access control based on execution history. In *Proceedings of the 10th Annual Network and Distributed System Symposium*, February 2003.

[4] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the Java language. In *Object Oriented Programing: Systems, Languages, and Applications (OOPSLA)*, October 1997.

[5] Bowen Alpern and Fred Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1987.

[6] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[7] Apache Software Foundation. *Byte Code Engineering Library*, 2003. `http://jakarta.apache.org/bcel/`.

[8] Lujo Bauer, Andrew W. Appel, and Edward W. Felten. Mechanisms for secure modular programming in Java. *Software—Practice and Experience*, 33(5):461–480, 2003.

[9] Lujo Bauer, Jarred Ligatti, and David Walker. More enforceable security policies. In *Foundations of Computer Security*, Copenhagen, Denmark, July 2002.

[10] Lujo Bauer, Jarred Ligatti, and David Walker. Types and effects for non-interfering program monitors. In M. Okada, B. Pierce, A. Scedrov, H. Tokuda, and A. Yonezawa, editors, *Software Security—Theories and Systems. Mext-NSF-JSPS International Symposium, ISSS 2002, Tokyo, Japan, November 8-10, 2002, Revised Papers*, volume 2609 of *Lecture Notes in Computer Science*. Springer, 2003.

[11] Lujo Bauer, Jay Ligatti, and David Walker. Composing security policies with polymer. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, Chicago, June 2005.

[12] Lujo Bauer, Jay Ligatti, and David Walker. Polymer: A language for composing run-time security policies, 2005. `http://www.cs.princeton.edu/sip/projects/polymer/`.

[13] K. J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, MITRE Corporation, July 1975.

[14] D. F. C. Brewer and M. J. Nash. The chinese wall security policy. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 206–214, 1989.

[15] J. R. Büchi. On a decision method in restricted second order arithmetic. In *Proceedings of the 1960 International Congress on Logic, Methodology, and Philosophy of Science*, pages 1–11, Stanford, 1962.

[16] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The Ponder policy specification language. *Lecture Notes in Computer Science*, 1995:18–39, 2001.

[17] Guy Edjlali, Anurag Acharya, and Vipin Chaudhary. History-based access control for mobile code. In *ACM Conference on Computer and Communications Security*, pages 38–48, 1998.

[18] Úlfar Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Cornell University, November 2003.

[19] Úlfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the New Security Paradigms Workshop*, pages 87–95, Caledon Hills, Canada, September 1999.

[20] Úlfar Erlingsson and Fred B. Schneider. IRM enforcement of Java stack inspection. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2000.

[21] David Evans. *Policy-directed Code Safety*. PhD thesis, Massachusetts Institute of Technology, February 2000.

[22] David Evans and Andrew Twyman. Flexible policy-directed code safety. In *IEEE Security and Privacy*, Oakland, CA, May 1999.

[23] Philip W. L. Fong. Access control by tracking shallow execution history. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2004.

[24] Kevin Hamlen, Greg Morrisett, and Fred B. Schneider. Computability classes for enforcement mechanisms. Technical Report 2003-1908, Cornell University, October 2003. A revised version will appear in *Transactions on Programming Languages and Systems (TOPLAS)*.

[25] Klaus Havelund and Grigore Roşu. Efficient monitoring of safety properties. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(2):158–173, August 2004.

[26] Benjamin Hindman and Dan Grossman. Strong atomicity for java without virtual-machine support. March 2006.

[27] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java. In *ACM conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 132–146, Denver, CO, August 1999.

[28] Clinton Jeffery, Wenyi Zhou, Kevin Templer, and Michael Brazell. A lightweight architecture for program execution monitoring. In *Program Analysis for Software Tools and Engineering (PASTE)*, pages 67–74. ACM Press, 1998.

[29] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, Monterey, CA, June 2002.

[30] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. An overview of AspectJ. In *European Conference on Object-oriented Programming*. Springer-Verlag, 2001.

[31] Gregor Kiczales, John Irwin, John Lamping, Jean-Marc Loingtier, Cristina Videria Lopes, Chris Maeda, and Anurag Mendhekar. Aspect-oriented programming. *ACM Comput. Surv.*, 28(4es):154, 1996.

[32] Moonjoo Kim, Sampath Kannan, Insup Lee, Oleg Sokolsky, and Mahesh Viswantathan. Computational analysis of run-time monitoring—fundamentals of Java-MaC. In *Run-time Verification*, June 2002.

[33] Moonjoo Kim, Mahesh Viswanathan, Hanene Ben-Abdallah, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Formally specified monitoring of temporal properties. In *European Conference on Real-time Systems*, York, UK, June 1999.

[34] Padmanabhan Krishnan. A monitoring policy calculus. Technical Report CSA-05-01, Bond University, 2005.

[35] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions of Software Engineering*, 3(2):125–143, 1977.

[36] Yingsha Liao and Donald Cohen. A specificational approach to high level program monitoring and measuring. *IEEE Trans. Softw. Eng.*, 18(11):969–978, 1992.

[37] Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: Enforcement mechanisms for run-time security policies. Technical Report TR-681-03, Princeton University, May 2003.

[38] Jay 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.

[39] Jay Ligatti, Lujo Bauer, and David Walker. Enforcing non-safety security policies with program monitors. In *10th European Symposium on Research in Computer Security (ESORICS)*, Milan, Italy, September 2005.

[40] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the 6th annual ACM Symposium on Principles of Distributed Computing*, pages 137–151. ACM Press, 1987.

[41] Gary McGraw and Edward W. Felten. *Securing Java: Getting Down to Business with Mobile Code*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[42] Robin Milner, Mads Tofte, Robert Harper, and Dave MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[43] George C. Necula, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In *Symposium on Principles of Programming Languages*, pages 128–139, 2002.

[44] William H. Paxton. A client-based transaction system to maintain data integrity. In *Proceedings of the 7th ACM symposium on Operating Systems Principles*, pages 18–23. ACM Press, 1979.

[45] Allen Petersen. Pooka: A Java email client, 2003. `http://www.suberic.net/pooka/`.

[46] W. Robinson. Monitoring software requirements using instrumented code. In *HICSS '02: Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS'02)-Volume 9*, page 276.2, Washington, DC, USA, 2002. IEEE Computer Society.

[47] Mark Russinovich. Sony, rootkits and digital rights management gone too far, October 2005. http://www.sysinternals.com/blog/2005/10/sony-rootkits-and-digital-rights.html.

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

[49] K. Sen, A. Vardhan, G. Agha, and G. Rosu. Efficient decentralized monitoring of safety in distributed systems. In *26th International Conference on Software Engineering (ICSE'04)*, pages 418–427, 2004.

[50] David B. Tucker and Shriram Krishnamurthi. Pointcuts and advice in higher-order languages. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, pages 158–167, 2003.

[51] Mahesh Viswanathan. *Foundations for the Run-time Analysis of Software Systems*. PhD thesis, University of Pennsylvania, 2000.

[52] Robert Wahbe, Steven Lucco, Thomas Anderson, and Susan Graham. Efficient software-based fault isolation. In *Fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, Asheville, December 1993.

[53] David Walker, Steve Zdancewic, and Jay Ligatti. A theory of aspects. In *ACM International Conference on Functional Programming*, Uppsala, Sweden, August 2003.