Computability Classes for Enforcement Mechanisms*

KEVIN W. HAMLEN
Cornell University
GREG MORRISETT
Harvard University
and
FRED B. SCHNEIDER
Cornell University

A precise characterization of those security policies enforceable by program rewriting is given. This also exposes and rectifies problems in prior work, yielding a better characterization of those security policies enforceable by execution monitors as well as a taxonomy of enforceable security policies. Some but not all classes can be identified with known classes from computational complexity theory.

Categories and Subject Descriptors: D.1.2 [**Programming Techniques**]: Automatic Programming; D.2.1 [**Software Engineering**]: Requirements/Specifications; D.4.6 [**Operating Systems**]: Security and Protection—*Access controls*; F.1.1 [**Computation By Abstract Devices**]: Models of Computation—*Automata*; *Bounded-action devices*; F.1.3 [**Computation By Abstract Devices**]: Complexity Classes—*Complexity hierarchies*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Specification techniques*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms: Security

Additional Key Words and Phrases: program rewriting, reference monitors, execution monitoring, inlined reference monitoring, security automata, edit automata

1. INTRODUCTION

Extensible systems, such as web browsers which download and run applet programs, or operating systems which incorporate drivers for new devices, must ensure that their extensions behave in a manner consistent with the intentions of the system

*Supported in part by AFOSR grants F49620-00-1-0198 and F49620-03-1-0156, Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory Air Force Material Command USAF under agreement number F30602-99-1-0533, National Science Foundation Grant 9703470, ONR Grant N00014-01-1-0968, and a grant from Intel Corporation. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government.
Authors' Addresses: Kevin W. Hamlen and Fred B. Schneider; Department of Computer Science, Cornell University; Ithaca, New York 14853. Greg Morrisett; Division of Engineering and Applied Science, Harvard University; Cambridge, MA 02138.

designer and its users. When unacceptable behavior goes unchecked, damage can result—not only to the system itself but also to connected systems.

Security enforcement mechanisms are employed to prevent unacceptable behavior. Recently, attention has turned to formally characterizing types of enforcement mechanisms and identifying the classes of security policies they can enforce [Ligatti et al. 2003; Schneider 2000; Viswanathan 2000]. This work allows us to assess the power of different mechanisms, choose mechanisms well suited to particular security needs, identify kinds of attacks that might still succeed even after a given mechanism has been deployed, and derive meaningful completeness results for newly developed mechanisms.

The abstract model for security policies developed by Schneider [2000] and refined by Viswanathan [2000] characterizes a class of policies meant to capture what could be effectively enforced through execution monitoring. *Execution monitors* are enforcement mechanisms that work by monitoring the computational steps of untrusted programs and intervening whenever execution is about to violate the security policy being enforced. Execution monitoring, however, can be viewed as an instance of the more general technique of *program-rewriting*, wherein the enforcement mechanism transforms untrusted programs before they are executed so as to render them incapable of violating the security policy to be enforced. No characterization of the class of security policies enforceable by program-rewriting has been developed (to our knowledge). Since numerous systems [Deutsch and Grant 1971; Small 1997; Wahbe et al. 1993; Erlingsson and Schneider 2000a; Evans and Twynman 1999; Erlingsson and Schneider 2000b] use program-rewriting in ways that go beyond what can be modeled as an execution monitor, a characterization of the class of policies enforceable by program-rewriters would be useful. So we here extend Schneider's and Viswanathan's model to characterize this new class of policies, the *RW-enforceable* policies, corresponding to what can be effectively enforced through program-rewriting.

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

We proceed as follows. We establish a formal model of security enforcement in §2. Next, in §3 we use that model to characterize and relate three methods of security enforcement: static analysis, execution monitoring, and program rewriting. Using the results of these analyses, §4 exposes and corrects flaws in prior work that cause Schneider's and Viswanathan's class to admit policies not enforceable by any execution monitor. Related and future work is discussed in §5. Finally, §6 summarizes the results of the prior sections.

2. FORMAL MODEL OF SECURITY ENFORCEMENT

2.1 Programs and Executions

An *enforcement mechanism* prevents unacceptable behavior by *untrusted programs*. Fundamental limits on what an enforcement mechanism can prevent arise whenever that mechanism is built using computational systems no more powerful than the systems upon which the untrusted programs themselves are based, because the incompleteness results of Gödel [1931] and Turing [1936] then imply there will be questions about untrusted programs unanswerable by the enforcement mechanism.

To expose these unanswerable questions, untrusted programs must be represented using some model of computation. The Turing Machine (TM) [Turing 1936] is an obvious candidate because it is well understood and because a wide range of security policies can be encoded as properties of Turing Machines [Marcus 1989; Harrison et al. 1976].

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

However, there are two reasons that the traditional definition of a TM, as a one-tape finite state machine that accepts or rejects finite-length input strings, is unsuitable for representing untrusted programs. First, it does not model non-terminating programs well. Operating systems, which are programs intended to run indefinitely, are not easily characterized in terms of acceptance or rejection of a finite input. Second, the traditional definition of a TM does not easily distinguish runtime information that is observable by the outside world (e.g. by an enforcement mechanism) from runtime information that is not observable because all runtime information is typically encoded on a single tape. Any realistic model of execution monitoring must be rich enough to express the monitor's limited power to access some but not all information about the untrusted program as it runs.

Therefore, untrusted programs are modeled in this paper by a multitape variant of a traditional TM (multitape TM's being equivalent in computational power to single-tape TM's [Hopcroft and Ullman 1979, p. 161]) that we term a *program machine* (PM). PM's are deterministic TM's (i.e. TM's with deterministic transition relations) that manipulate three infinite-length tapes:

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

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

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

mechanism.

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

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

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

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

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

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

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

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

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

A weaker set of assumptions about E that permits enforcement mechanisms access to less information, but that still captures the predictive power of interesting enforcement mechanisms might be possible but is left as future work. Independent work on this problem [Fong 2004] is discussed in §5.

Following Schneider [2000], program *executions* are modeled as sequences χ of events from E. Without loss of generality, we assume that complete executions are always infinite event sequences. (If an untrusted program's termination is an observable event, then it can be modeled by a PM that loops, repeatedly exhibiting a distinguished event e_{end}, instead of terminating. If program termination is not observable, E can be augmented with an event e_{skip} that indicates that either the untrusted program has terminated or that no security-relevant event has taken place on a particular computational step.) Many of our analyses will involve both complete executions and their finite prefixes, and we use χ to refer to both infinite and finite event sequences unless explicitly stated otherwise. Each finite prefix of an execution encodes the information available to the enforcement mechanism up to that point in the execution. Define $|\cdot| : E^\omega \to (\mathbb{N} + \{\infty\})$ such that $|\chi|$ is the length of sequence χ if χ is finite, and $|\chi|$ is ∞ if χ is infinite. When $i \in \mathbb{N}$ and $1 \leq i \leq |\chi|$ then let $\chi[i]$ denote the ith event of χ, let $\chi[..i]$ denote the length-i prefix of χ, and let $\chi[i+1..]$ denote the suffix of χ consisting of all but the first i events.

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

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

[1] This assumption might appear to give an enforcement mechanism arbitrarily powerful decision-making ability, but we will see in §3 that the power is still quite limited because unrestricted access to the program text is tempered by time limits on the use of that information.

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

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

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

2.2 Security Policies

A *security policy* defines a binary partition on the set of all (computable) sets of executions. Each (computable) set of executions corresponds to a PM, so a security policy divides the set of all PM's into those that *satisfy* the policy and those that do not. This definition of security policies is broad enough to express most things usually considered security policies [Schneider 2000], including

— *access control* policies, which are defined in terms of a program's behavior on an arbitrary individual execution for an arbitrary finite period,

— *availability* policies, which are defined in terms of a program's behavior on an arbitrary individual execution over an infinite period, and

— *information flow* policies, which are defined in terms of the set of all executions—and not the individual executions in isolation—that a program could possibly exhibit.

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

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

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

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

That is, a PM M satisfies a security policy $\mathcal{P}$ if and only if all possible executions of M satisfy predicate $\hat{\mathcal{P}}$. $\hat{\mathcal{P}}$ is called a *detector* for $\mathcal{P}$. For example, if we define a detector $\hat{\mathcal{P}}_{boot}(\chi)$ to hold exactly when χ does not contain any event $e_{writes(i)}$ for $0 \leq i < 512$, then policy $\mathcal{P}_{boot}$ (above) is the policy induced by detector $\hat{\mathcal{P}}_{boot}$.

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

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

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

3. MODELING VARIOUS SECURITY ENFORCEMENT MECHANISMS

The framework defined in §2 can be used to model many security enforcement mechanisms, including static analyses (c.f. [Lindholm and Yellin 1999, Chapter 4.9.1, Passes 1-3; Morrisett et al. 1999; Myers 1999; Nachenberg 1997]), reference monitors (c.f. [Anderson 1972; Lampson 1971; Lindholm and Yellin 1999, Chapter 4.9.1, Pass 4; Rees and Clinger 1991; Moonjoo Kim and Sokolsky 2001]), and program rewriters (c.f. [Deutsch and Grant 1971; Erlingsson and Schneider 2000a; Erlingsson and Schneider 2000b; Evans and Twynman 1999; Small 1997; Wahbe et al. 1993]).

3.1 Static Analysis

Enforcement mechanisms that accept or reject an untrusted program strictly prior to running the untrusted program are termed *static analyses*. Here, the enforcement mechanism must determine whether the untrusted program satisfies the security policy within a finite period of time.[4] Accepted programs are permitted to run unhindered; rejected programs are not run at all. Examples of static analyses include static type-checkers for type-safe languages, like that of the Java Virtual Machine[5] [Lindholm and Yellin 1999, Chapter 4.9.1, Passes 1-3] and TAL [Morrisett et al. 1999]. JFlow [Myers 1999] and others use static analyses to provide guarantees about other security policies such as information flow. Standard virus scanners [Nachenberg 1997] also implement static analyses.

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

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

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

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

Formally, a security policy $\mathcal{P}$ is deemed *statically enforceable* in our model if there exists a TM $M_{\mathcal{P}}$ that takes an encoding of an arbitrary PM M as input and, if $\mathcal{P}(M)$ holds, then $M_{\mathcal{P}}(M)$ accepts in finite time; otherwise $M_{\mathcal{P}}(M)$ rejects in finite time. Thus, by definition, statically enforceable security policies are the recursively decidable properties of TM's:

THEOREM 3.1.1. *The class of statically enforceable security policies is the class of recursively decidable properties of programs (also known as class* Π_0 *of the arithmetic hierarchy).*

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

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

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

3.2 Execution Monitoring

Reference monitors [Anderson 1972; Ware 1979] and other enforcement mechanisms that operate alongside an untrusted program are termed *execution monitors* (EM's) in [Schneider 2000]. An EM intercepts security-relevant events exhibited as the untrusted program executes, and the EM intervenes upon seeing an event that would lead to a violation of the policy being enforced. The intervention might involve terminating the untrusted program or might involve taking some other corrective action.[6] Examples of EM enforcement mechanisms include access control list and capability-based implementations of access control matrices [Lampson 1971] as well as hardware support for memory protection. Runtime linking performed by the Java Virtual Machine [Lindholm and Yellin 1999, Chapter 4.9.1, Pass 4], and runtime type-checking such as that employed by dynamically typed languages like Scheme

[6]Schneider [2000] assumes the only intervention action available to an EM is termination of the untrusted program. Since we are concerned here with a characterization of what policies an EM can enforce, it becomes sensible to consider a larger set of interventions.

[Rees and Clinger 1991], are other examples. The MaC system [Moonjoo Kim and Sokolsky 2001] implements EM's through a combination of runtime event-checking and program instrumentation.

Schneider [2000] observes that for every EM-enforceable security policy $\mathcal{P}$ there will exist a detector $\hat{\mathcal{P}}$ such that

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

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

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

and thus the EM-enforceable policies are a subset of the safety properties[7].

Viswanathan [2000] observes that $\hat{\mathcal{P}}$ must also be computable (something left implicit in [Schneider 2000]), giving rise to a fourth restriction:

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

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

EM4 is critical because it rules out detectors that are arbitrarily powerful and thus not available to any real EM implementation. For example, the policy that a PM must eventually halt—a liveness property that no EM can enforce [Lamport 1977; Schneider 2000]—satisfies EM1 – EM3 but not EM4.

In §4.1 we show that real EM's are limited by additional constraints. However, class EM_{orig} constitutes a useful upper bound on the set of policies enforceable by execution monitors. Viswanathan [2000] shows that EM_{orig} is equivalent to the co-recursively enumerable (coRE) properties, also known as class Π_1 of the arithmetic hierarchy. A security policy $\mathcal{P}$ is coRE when there exists a TM $M_{\mathcal{P}}$ that takes an arbitrary PM M as input and rejects it in finite time if $\neg\mathcal{P}(M)$ holds; otherwise $M_{\mathcal{P}}(M)$ loops forever. The equivalence of EM_{orig} to the coRE properties will be used throughout the remainder of the paper. Since Viswanathan's model differs substantially from ours, we reprove this result for our model.

THEOREM 3.2.1. *The class given by EM1 – EM4 is the class of co-recursively enumerable (coRE) properties of programs (also known as the* Π_1 *class of the arithmetic hierarchy).*

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

By EM1, $\mathcal{P}(M) \equiv (\forall \chi : \chi \in X_M : \hat{\mathcal{P}}(\chi))$ for some $\hat{\mathcal{P}}$ satisfying EM2 – EM4. EM4 guarantees that a TM can decide $\hat{\mathcal{P}}(\chi)$ for finite χ. We can therefore construct $M_{\mathcal{P}}$, as follows: When given M as input, $M_{\mathcal{P}}$ begins to iterate through every finite prefix

[7] A *safety property* is a property that stipulates some "bad thing" does not happen during execution [Lamport 1977].

χ of executions in X_M. For each, it decides $\hat{\mathcal{P}}(\chi)$. If it finds a χ such that $\neg\hat{\mathcal{P}}(\chi)$ holds, it halts. (This is possible because EM4 guarantees that $\hat{\mathcal{P}}(\chi)$ is recursively decidable.) Otherwise it continues iterating indefinitely.

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

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

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

Since the coRE properties are a proper superset of the recursively decidable properties [Papadimitriou 1995, p. 63], every statically enforceable policy is trivially enforceable by an EM—the static analysis would be performed by the EM immediately after the PM exhibits its first event (i.e., immediately after the program is loaded). Statically enforceable policies are guaranteed to be computable in a finite period of time, so the EM will always be able to perform this check in finite time and terminate the untrusted PM if the check fails.

Even though the power of the EM approach is strictly greater than that of the static approach, this does not mean that the EM approach is to be universally preferred over the static approach. Depending on the policy to be enforced, either approach might be preferable to the other for engineering reasons. For example, static enforcement mechanisms predict policy violations before a program is run and therefore do not slow the program, whereas EM's usually slow execution due to their added runtime checks. Also, an EM might signal security policy violations arbitrarily late into an execution and only on some executions, whereas a static analysis reveals prior to execution whether that program could violate the policy. Thus, recovering from policy violations discovered by an EM can be more difficult than recovering from those discovered by a static analysis. In particular, an EM might need to roll back a partially completed computation, whereas a static analysis always discovers the violation before computation begins. Alternatively, some policies, though enforceable by a static analysis, are simpler to enforce with an EM, reducing the risk of implementation errors in the enforcement mechanism. Moreover, the comparison of static enforcement to EM-enforcement given in Theorem 3.2.1 assumes that both are being given the same information (because in our model an EM can examine the untrusted program via event e_M). If a static analysis is provided one representation of the program (e.g., source code) and an EM

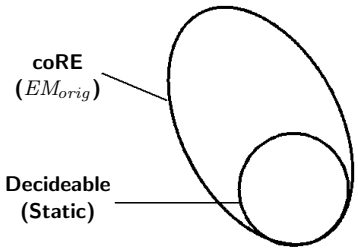


Fig. 1. The relationship of the static to the coRE policies.

is provided another in which some of the information has been erased (e.g., object code), then each might well be able to enforce policies that the other cannot.

Theorem 3.2.1 also suggests that there are policies enforceable by EM's that are not statically enforceable, since $\Pi_0 \subset \Pi_1$. Policy $\mathcal{P}_{boot}$ given in §2.2 is an example. More generally, assuring that a PM will never exhibit some prohibited event is equivalent to solving the Halting Problem, which is known to be undecidable and therefore is not statically enforceable. EM's enforce such "undecidable" security policies by waiting until a prohibited event is about to occur, and then signaling the violation.

The relationship of the static policies to class EM_{orig} is depicted in Figure 1.

3.3 Program Rewriting and RW$_{\approx}$-Enforceable Policies

An alternative to static analysis or execution monitoring is *program rewriting*. A *program-rewriter* modifies, in finite time, untrusted programs prior to their execution. Use of program rewriting for enforcing security policies dates back at least to 1969 [Deutsch and Grant 1971]. More recently, we find program rewriting being employed in software-based fault-isolation (SFI) [Small 1997; Wahbe et al. 1993] as a way of implementing memory-safety policies, and in PSLang/PoET [Erlingsson and Schneider 2000a] and Naccio [Evans and Twynman 1999] for enforcing security policies in Java. Program-rewriters can also be seen as a generalization of execution monitoring, since they can be used to implement an EM as an *in-lined reference monitor* (IRM) whereby an EM is embedded into the untrusted program [Erlingsson and Schneider 2000b]. The approach is appealing, powerful, and quite practical, so understanding what policies it can enforce is a worthwhile goal.

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

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

Given a PM-equivalence relation, we define a policy $\mathcal{P}$ to be *RW$_{\approx}$-enforceable* if there exists a total, computable *rewriter function* $R : PM \to PM$ such that for all

PM's M,

$$\mathcal{P}(R(M)) \tag{RW1}$$
$$\mathcal{P}(M) \Longrightarrow M \approx R(M) \tag{RW2}$$

Thus, for a security policy $\mathcal{P}$ to be considered $\text{RW}_{\approx}$-enforceable, there must exist a way to transform a PM so that the result is guaranteed to satisfy $\mathcal{P}$ (RW1) and if the original PM already satisfied $\mathcal{P}$, then the transformed PM is equivalent (but not necessarily identical) to the old (RW2). It is important to note that RW1 precludes a program-rewriter from producing as output any PM that does not satisfy the policy. As we shall see in §4, this requirement leads to a subtle but important distinction between the class of $\text{RW}_{\approx}$-enforceable policies and EM_{orig}.

Equivalence relation $\approx_\chi$ in PM-equivalence is defined independently of any security policy, but the choice of any particular $\approx_\chi$ places an implicit constraint on which detectors can be considered in any of our analyses of policies enforceable by detectors. In particular, it is sensible to consider only those detectors $\hat{\mathcal{P}}$ that satisfy

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

Such a detector will be said to be *consistent* with $\approx_\chi$ since it never classifies one execution as acceptable and another as unacceptable when the two are equivalent according to $\approx_\chi$. Program rewriters presume equivalent executions are interchangeable, which obviously isn't the case if one execution is acceptable and the other is not. Thus, detectors that are not consistent with $\approx_\chi$ are not compatible with the model. In an analysis of any particular enforcement mechanism involving detectors, $\approx_\chi$ should be defined in such a way that all detectors supported by the mechanism are consistent with $\approx_\chi$, and are therefore covered by the analysis.

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

THEOREM 3.3.1. *Every satisfiable, statically enforceable policy is $RW_{\approx}$-enforceable.*

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

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

R is total because it assigns a TM to every M, and it is computable because $\mathcal{P}$ is statically enforceable and therefore, by Theorem 3.1.1, recursively decidable. R satisfies RW1 because its range is restricted to programs that satisfy $\mathcal{P}$. Finally, R

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

satisfies RW2 because whenever $\mathcal{P}(M)$ holds, $R(M) = M$. Thus $M \approx R(M)$ holds because $M \approx M$ holds by the reflexivity of equivalence relations. We conclude that $\mathcal{P}$ is $\text{RW}_{\approx}$-enforceable. □

Theorems 3.2.1 and 3.3.1 together imply that the intersection of class EM_{orig} with the $\text{RW}_{\approx}$-enforceable policies includes all satisfiable, statically enforceable policies.

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

THEOREM 3.3.2. *Assume that PM-equivalence relation $\approx$ is recursively decidable, and let $\hat{\mathcal{P}}$ be a detector consistent with $\approx_\chi$. If the policy $\mathcal{P}$ induced by $\hat{\mathcal{P}}$ is $\text{RW}_{\approx}$-enforceable then $\mathcal{P}$ is statically enforceable.*

PROOF. Exhibit a finite procedure for deciding $\mathcal{P}$, thereby establishing that $\mathcal{P}$ is statically enforceable by Theorem 3.1.1.

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

$\mathcal{P}(R(M))$	(RW1)	
$(\forall \chi : \chi \in X_{R(M)} : \hat{\mathcal{P}}(\chi))$	($\hat{\mathcal{P}}$ induces $\mathcal{P}$)	(1)
$(\forall \sigma : \sigma \in \Gamma^\omega : \chi_{M(\sigma)} \approx_\chi \chi_{R(M)(\sigma)})$	(because $M \approx R(M)$)	
$(\forall \sigma : \sigma \in \Gamma^\omega : \hat{\mathcal{P}}(\chi_{M(\sigma)}) \equiv \hat{\mathcal{P}}(\chi_{R(M)(\sigma)}))$	(consistency)	(2)
$(\forall \sigma : \sigma \in \Gamma^\omega : \hat{\mathcal{P}}(\chi_{M(\sigma)}))$	(by 1 and 2)	(3)
$\mathcal{P}(M)$	(by 3)	

Thus, $\mathcal{P}(M)$ has been decided in finite time and we conclude by Theorem 3.1.1 that $\mathcal{P}$ is statically enforceable. □

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

If PM-equivalence is not recursively decidable, then there exist policies that are $\text{RW}_{\approx}$-enforceable but not statically enforceable. $\mathcal{P}_{boot}$ of §2.2, is an example. A rewriting function can enforce $\mathcal{P}_{boot}$ by taking a PM M as input and returning a new PM M' that is exactly like M except just before every computational step of M, M' simulates M for one step into the future on every possible input symbol to see if M will exhibit any prohibited event $\{e_i | 0 \leq i < 512\}$. If any prohibited event is exhibited, then M' is terminated immediately; otherwise the next computational step is performed.

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

THEOREM 3.3.3. *There exist non-coRE PM-equivalence relations $\approx$ and policies that are $RW_{\approx}$-enforceable but not coRE.*

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

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

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

That is, M_{EQ} treats its input tape as an encoding of a pair of PM's M_1 and M_2, and builds a new PM M_3 that is exactly like the universal PM M_U except that it simulates M_1 when it receives M_2 as input instead of simulating M_2. PM M_{EQ} then queries oracle O_U with M_3. Observe that $M_{EQ} \approx^{TM} M_U$ iff $M_1 \approx^{TM} M_2$. Thus, $M_{EQ}(M_1, M_2)$ holds iff $M_1 \approx^{TM} M_2$. □

The proof of Theorem 3.3.3 gives a simple but practically uninteresting example of an $\text{RW}_{\approx}$-enforceable policy that is not coRE. Examples of non-coRE, $\text{RW}_{\approx}$-enforceable policies having practical significance do exist. Here is one.

The Secret File Policy: Consider a filesystem that stores a file whose existence should be kept secret from untrusted programs. Suppose untrusted programs have an operation for retrieving a listing of the directory that contains the secret file. System administrators wish to enforce a policy that prevents the existence of the secret file from being leaked to untrusted programs. So, an untrusted PM satisfies the "secret file policy" if and only if the behavior of

the PM is identical to what its behavior would be if the secret file were not stored in the directory.

The policy in this example is not in coRE because deciding whether an arbitrary PM has equivalent behavior on two arbitrary inputs is as hard as deciding whether two arbitrary PM's are equivalent on all inputs. And recall that PM-equivalence ($\approx$) is not coRE. Thus an EM cannot enforce this policy.[9] However, this policy can be enforced by program rewriting, because a rewriting function never needs to explicitly decide if the policy has been violated in order to enforce the policy. In particular, a rewriter function can make modifications that do not change the behavior of programs that satisfy the policy, but do make safe those programs that don't satisfy the policy. For the example above, program rewriting could change the untrusted program so that any attempt to retrieve the contents listing of the directory containing the secret file yields an abbreviated listing that excludes the secret file. If the original program would have ignored the existence of the file, then its behavior is unchanged. But if it would have reacted to the secret file, then it no longer does.

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

THEOREM 3.3.4. *There exist non-coRE PM-equivalence relations $\approx$ and policies that are not RW$_{\approx}$-enforceable.*

PROOF. Define policy $\mathcal{P}_{NE}(M) =_{\text{def}} \left(\exists \sigma : \sigma \in \Gamma^{\omega} : M(\sigma) \text{ halts}\right)$ and define $\approx^{TM}$ as in the proof of Theorem 3.3.3 (i.e. $\approx^{TM}$ is defined to be language-equivalence for TM's). Observe that policy $\mathcal{P}_{NE}$ is RE but not recursively decidable. We show that if $\mathcal{P}_{NE}$ were RW$_{\approx^{TM}}$-enforceable, then it would be recursively decidable—a contradiction.

Expecting a contradiction, assume that $\mathcal{P}_{NE}$ is RW$_{\approx^{TM}}$-enforceable. Then there exists a rewriting function R satisfying RW1 and RW2. Use R to decide $\mathcal{P}_{NE}$ in the following way: Let an arbitrary PM M be given. To decide if $\mathcal{P}_{NE}(M)$ holds, construct a new PM M' that treats its input tape as a positive integer i followed by a string σ. (For example, if the input alphabet $\Gamma = \{0, 1\}$, then $i\sigma$ might be encoded as $1^i 0\sigma$. Thus, every possible input string represents some valid encoding of an integer-string pair, and every integer-string pair has some valid encoding.) Upon receiving $i\sigma$ as input, PM M' simulates $M(\sigma)$ for i steps and halts if $M(\sigma)$ halts in that time; otherwise M' loops. Next, compute $R(M')$. Since R satisfies RW1, there exists a string on which $R(M')$ halts. Simulate $R(M')$ on each possible input string for larger and larger numbers of steps until such a string $j\sigma'$ is found. Next, simulate $M(\sigma')$ for j steps. We claim that $\mathcal{P}_{NE}(M)$ holds iff $M(\sigma')$ halts in j steps. If $M(\sigma')$ halts in j steps, then obviously $\mathcal{P}_{NE}(M)$ holds. If $M(\sigma')$ does not halt in j steps, then $M'(j\sigma')$ loops by construction. Since $M'(j\sigma')$ loops but

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

$R(M')(j\sigma')$ halts, it follows that $R(M') \not\approx^{TM} M'$. By RW2, $\mathcal{P}_{NE}(M)$ therefore does not hold.

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

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

THEOREM 3.3.5. *There exist non-coRE PM-equivalence relations $\approx$ such that the class of $RW_{\approx}$-enforceable policies is not equivalent to any class of the arithmetic hierarchy.*

PROOF. The proof of Theorem 3.3.3 showed that the Π_2-hard policy $\mathcal{P}_U$ is $\mathrm{RW}_{\approx^{TM}}$-enforceable. Theorem 3.3.4 showed that the RE policy $\mathcal{P}_{NE}$ is not $\mathrm{RW}_{\approx^{TM}}$-enforceable. Since Π_2 is a superclass of RE, there is no class of the arithmetic hierarchy that includes $\mathcal{P}_U$ but not $\mathcal{P}_{NE}$. We conclude that the class of $\mathrm{RW}_{\approx^{TM}}$-enforceable policies is not equivalent to any class of the arithmetic hierarchy. □

4. EXECUTION MONITORS AS PROGRAM REWRITERS

Since EM's can be implemented by program-rewriters as in-lined reference monitors [Erlingsson and Schneider 2000a], one might expect EM_{orig} to be a subclass of the $\mathrm{RW}_{\approx}$-enforceable policies. However, in this section we show that there are policies in EM_{orig} that are not $\mathrm{RW}_{\approx}$-enforceable. In section §4.1 we identify some of these policies and argue that they cannot actually be enforced by EM's either. Thus, EM_{orig} is a superclass of the class of policies that EM's can enforce. In §4.2 we then show how the definition of $\mathrm{RW}_{\approx}$-enforceable policies presented in §3.3 can be leveraged to obtain a precise characterization of the EM-enforceable policies.

4.1 EM-enforceable Policies

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

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

For example, [Schneider 2000] presumes that an EM intervenes only by terminating the PM. Define e_{end} to be an event that corresponds to termination of the PM. The policy $\mathcal{P}_{\{e_{end}\}}$, which demands that no PM may terminate, is not enforceable by such an EM even though it satisfies the definition of EM_{orig}. In addition, more complex policies involving e_{end}, such as the policy that demands that every PM must exhibit event e_1 before it terminates (i.e. before it exhibits event e_{end}) are also unenforceable by such an EM, even though they too are members of EM_{orig}. The power of an EM is thus limited by the set of interventions available to it, in addition to the limitations described by the definition of EM_{orig}.

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

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

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

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

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

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

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

However, if an EM were to use detector $\hat{\mathcal{P}}_{RU2}$ to enforce policy $\mathcal{P}_{RU}$, it would not be able to prevent two untrusted programs from simultaneously using the resource. An EM using detector $\hat{\mathcal{P}}_{RU2}$ would only discover that an execution should be rejected when the untrusted program attempts to exhibit e_{use} after having exhibited e_{rel} in the past. At that point the service might have already allocated the resource to another untrusted program and would not be able to revoke the resource from either program. Detector $\hat{\mathcal{P}}_{RU2}$ is therefore violated at a time too late to permit the enforcement mechanism to guarantee that all executions satisfy detector $\hat{\mathcal{P}}_{RU1}$. Violations of $\hat{\mathcal{P}}_{RU1}$ are detected by the EM but cannot be corrected.

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

The power of an EM derives from the collection of detectors it offers policy-writers. A small collection of detectors might be stretched to "enforce" all coRE policies according to the terms of EM_{orig}, but in doing this, some of those policies will be "enforced" in ways that permit bad events to occur, which could be unacceptable to those wishing to actually prevent those bad events. Proofs that argue that some real enforcement mechanism is capable of enforcing all policies in EM_{orig} are thus misleading. For example, the MaC system was shown to be capable of enforcing all coRE policies [Viswanathan 2000], but policies like $\mathcal{P}_{RU}$ cannot be enforced by MaC in such a way as to signal the violations specified by $\hat{\mathcal{P}}_{RU1}$ before the violation has already occurred.

In §4.2 we show that the intersection of class EM_{orig} with the $\text{RW}_{\approx}$-enforceable policies constitutes a more suitable characterization of the EM-enforceable policies than class EM_{orig} alone. This is because RW1 and RW2 impose constraints upon an EM's ability to intervene. For example, in a setting where EM's can intervene by suppressing events that would otherwise be exhibited by an untrusted PM, one might model such interventions by an event $e_{supp(e')}$ that is exhibited whenever an EM suppresses event e'. If EM's cannot suppress e_{use} events, one might wish to enforce policy $\mathcal{P}'_{RU}$ defined by

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

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

Characterizing the EM-enforceable policies as the intersection of class EM_{orig} with the $\text{RW}_{\approx}$-enforceable policies therefore allows us to express policies that regard the whole system rather than just the the part of the system that does not include the EM. That is, it allows us to reify the EM into the computation and consider policies that regard this new composite computation rather than just the

computation defined by the untrusted PM's behavior in isolation. A enforcement mechanism can only be said to "enforce" a policy if it neither allows any untrusted PM to violate the policy, nor itself violates the policy in the course of "enforcing" it.

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

4.2 Benevolent Enforcement of Execution Policies

To account for the additional restrictions upon EM's described in §4.1, it will be useful to identify those detectors for which there is some means to enforce the policies they induce without producing executions that violate the detector. Formally, we define a detector $\hat{\mathcal{P}}$ to be *benevolent* if there exists a decision procedure $M_{\hat{\mathcal{P}}}$ for finite executions such that for all PM's M:

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

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

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

Detector $\hat{\mathcal{P}}_{RU2}$ of §4.1 is an example of a benevolent detector. The decision procedure $M_{\hat{\mathcal{P}}_{RU2}}(\chi)$ would simply scan χ and would reject iff e_{use} was seen after e_{rel}. However, detector $\hat{\mathcal{P}}_{RU1}$ of §4.1 is an example of a detector that is not benevolent. It is not possible to discover in finite time whether there exists some extension of execution χ that includes event e_{use} (or, conservatively, whether any execution of $\langle\!\langle\chi\rangle\!\rangle$ has an e_{use} after an e_{rel}). Therefore no suitable decision procedure $M_{\hat{\mathcal{P}}_{RU1}}$ satisfying B1 and B2 exists.

Benevolent detectors can be implemented so as to prevent all policy violations without hindering policy-satisfying programs. In the next theorem, we prove that if a policy is both coRE and $\text{RW}_{\approx}$-enforceable for some equivalence relation $\approx$ that permits the sorts of program transformations that are typically performed by in-lined reference monitors, then *every* detector that induces that policy (and that is also consistent with $\approx$ and satisfies EM2) is benevolent. That is, no matter which detector might be desired for enforcing such a policy, there is a way to implement that detector so that all policy violations are prevented but all executions of policy-satisfying programs are permitted. Thus, the class of policies that are both coRE

and $\text{RW}_{\approx}$-enforceable constitutes a good characterization of the policies that are actually enforceable by an EM. Such policies can be enforced by an EM that is implemented as an in-lined reference monitor, whereas other coRE policies cannot be so implemented (because they are not $\text{RW}_{\approx}$-enforceable).

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

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

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

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

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

THEOREM 4.2.1. *Let a detector $\hat{\mathcal{P}}$ satisfying EM2 be given, and assume that $\hat{\mathcal{P}}$ is consistent with $\approx_{\chi}^{IRM}$. If the policy $\mathcal{P}$ induced by $\hat{\mathcal{P}}$ is $\text{RW}_{\approx^{IRM}}$-enforceable and satisfies EM1 – EM4, then $\hat{\mathcal{P}}$ is benevolent.*

PROOF. Define a decision procedure $M_{\hat{\mathcal{P}}}$ for $\hat{\mathcal{P}}$ and show that it satisfies B1 and B2. We define $M_{\hat{\mathcal{P}}}$ as follows: When $M_{\hat{\mathcal{P}}}$ receives a finite execution prefix χ as input, it iterates through each $i \geq 1$ and for each i, determines if $\chi \approx_{\chi}^{IRM} \chi_{R(\langle\!\langle \chi \rangle\!\rangle)(\sigma)}[..i]$, where R is the rewriter given by the $\text{RW}_{\approx^{IRM}}$-enforceability of $\mathcal{P}$ and σ is the string of input symbols recorded in the trace tape as being read during χ. Both of these executions are finite, so by EQ1 this can be determined in finite time. If the two executions are equivalent, then $M_{\hat{\mathcal{P}}}$ halts with acceptance. Otherwise $M_{\hat{\mathcal{P}}}$ simulates $M_{\mathcal{P}}(\langle\!\langle \chi \rangle\!\rangle)$ for i steps, where $M_{\mathcal{P}}$ is a TM that halts if and only if its input represents a PM that violates $\mathcal{P}$. Such a TM is guaranteed to exist because $\mathcal{P}$ satisfies EM1 – EM4 and is therefore coRE by Theorem 3.2.1. If $M_{\mathcal{P}}(\langle\!\langle \chi \rangle\!\rangle)$ halts in i steps, then $M_{\hat{\mathcal{P}}}$ halts with rejection. Otherwise $M_{\hat{\mathcal{P}}}$ continues with iteration $i + 1$.

First, we prove that $M_{\hat{\mathcal{P}}}$ always halts. Suppose $\neg\mathcal{P}(\langle\!\langle \chi \rangle\!\rangle)$ holds. Then $M_{\mathcal{P}}$ will eventually reach a sufficiently large i that $M_{\mathcal{P}}(\langle\!\langle \chi \rangle\!\rangle)$ will halt, and thus $M_{\hat{\mathcal{P}}}$ will halt.

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

Suppose instead that $\mathcal{P}(\langle\!\langle\chi\rangle\!\rangle)$ holds. Then by RW2, $\langle\!\langle\chi\rangle\!\rangle \approx^{IRM} R(\langle\!\langle\chi\rangle\!\rangle)$. Applying the definition of $\approx^{IRM}$, we see that $\chi_{\langle\!\langle\chi\rangle\!\rangle(\sigma)} \approx_{\chi}^{IRM} \chi_{R(\langle\!\langle\chi\rangle\!\rangle)(\sigma)}$. Recalling that χ is a finite prefix of $\chi_{\langle\!\langle\chi\rangle\!\rangle(\sigma)}$, observe that EQ2 implies that there exists a (sufficiently large) i such that $\chi \approx_{\chi}^{IRM} \chi_{R(\langle\!\langle\chi\rangle\!\rangle)(\sigma)}[..i]$. Thus $M_{\hat{\mathcal{P}}}$ will halt.

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

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

The existence of policies in EM_{orig} that are not $\text{RW}_{\approx}$-enforceable can now be shown by demonstrating that there exist coRE policies with detectors that satisfy EM2 but that are not benevolent. By Theorem 4.2.1, no such policy can be both coRE and $\text{RW}_{\approx^{IRM}}$-enforceable.

THEOREM 4.2.2. *There exist detectors $\hat{\mathcal{P}}$ and equivalence relations $\approx_{\chi}^{IRM}$ such that $\hat{\mathcal{P}}$ is consistent with $\approx_{\chi}^{IRM}$, $\hat{\mathcal{P}}$ satisfies EM2 and EM3, the induced policy defined by $\mathcal{P}(M) =_{def} (\forall\chi : \chi \in X_M : \hat{\mathcal{P}}(\chi))$ satisfies EM1 – EM4, and yet $\hat{\mathcal{P}}$ is not benevolent.*

PROOF. Define $\mathcal{P}_{\{e_{end}\}}$ as in §4.1 and define $\hat{\mathcal{P}}_{NT}(\chi) =_{\text{def}} \mathcal{P}_{\{e_{end}\}}(\langle\!\langle\chi\rangle\!\rangle)$. That is, an execution satisfies $\hat{\mathcal{P}}_{NT}$ if and only if it comes from a program that never halts on any input. Define $\approx_{\chi}^{IRM}$ to be the identity relation over executions, and observe that $\hat{\mathcal{P}}_{NT}$ is consistent with $\approx_{\chi}^{IRM}$. Detector $\hat{\mathcal{P}}_{NT}$ satisfies EM2 because for every program M, either all prefixes of all of that program's executions satisfy $\hat{\mathcal{P}}_{NT}$ or none of them do. $\hat{\mathcal{P}}_{NT}$ satisfies EM3 because if an execution falsifies $\hat{\mathcal{P}}_{NT}$, then every finite prefix of that execution falsifies it as well.

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

$$\begin{aligned}
\mathcal{P}_{NT}(M) &\equiv (\forall\chi : \chi \in X_M : \hat{\mathcal{P}}_{NT}(\chi)) \\
&\equiv (\forall\chi : \chi \in X_M : \mathcal{P}_{\{e_{end}\}}(\langle\!\langle\chi\rangle\!\rangle)) && \text{(by def of } \hat{\mathcal{P}}_{NT} \text{ above)} \\
&\equiv (\forall\chi : \chi \in X_M : \mathcal{P}_{\{e_{end}\}}(M)) && \text{(because } \chi \in X_M\text{)} \\
&\equiv \mathcal{P}_{\{e_{end}\}}(M) && \text{(by def of } \mathcal{P}_{\{e_{end}\}} \text{ in §4.1)}
\end{aligned}$$

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

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

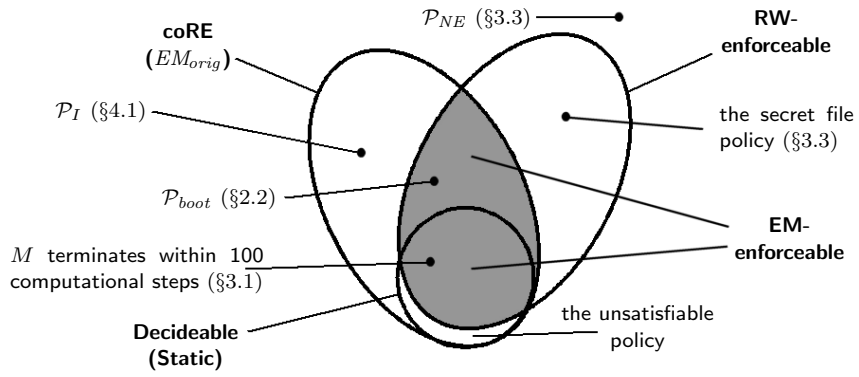


Fig. 2. Classes of security policies and some policies that lie within them.

To summarize, the relationship of the statically enforceable policies, class EM_{orig} (the coRE policies), and the policies enforceable by program rewriting (the $\text{RW}_{\approx}$-enforceable policies) is shown in Figure 2. The statically enforceable policies are a subset of the coRE policies and, with the exception of the unsatisfiable policy, a subset of the $\text{RW}_{\approx}$-enforceable policies. The shaded region indicates those policies that are both coRE and $\text{RW}_{\approx}$-enforceable. These are the policies that we characterize as EM-enforceable. There exist coRE policies outside this intersection, but all such policies are induced by some non-benevolent detector. Thus, using an EM to "enforce" any of these policies would result in program behavior that might continue to exhibit events that the policy was intended to prohibit. There are also $\text{RW}_{\approx}$-enforceable policies outside this intersection. These are policies that cannot be enforced by an EM but that can be enforced by a program rewriter that does not limit its rewriting to producing in-lined reference monitors.

Figure 2 also shows where various specific policies given throughout this paper lie within the taxonomy of policy classes. The policy "M terminates within 100 computational steps" given in §3.1 is an example of a policy that can be enforced by static analysis, execution monitoring, or program rewriting. Policy $\mathcal{P}_{boot}$, introduced in §2.2, is not enforceable by static analysis, but can be enforced by an EM or by a program rewriter. The secret file policy described in §3.3 is an example of a policy that cannot be enforced by any EM but that can be enforced by a program rewriter. Finally, policy $\mathcal{P}_I$ is one of the policies given in §4.1 that satisfies the definition of EM_{orig} but that cannot be enforced by any real EM in a way that prevents bad events from occurring on the system.

5. RELATED AND FUTURE WORK

Edit Automata. In contrast to program-rewriters, *edit automata* [Ligatti et al. 2003] modify executions rather than modifying programs. Cast in the framework of this paper, an edit automaton can intervene at each computational step by inhibiting any event a PM writes to its trace tape and/or writing additional events to the trace tape.

Like program-rewriters, the behavior of an edit automaton is constrained by an

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

To date, edit automata have been defined and analyzed only for a model in which all executions are assumed to be finite. In such a model, they can enforce any (computable) security policy by suppressing all events until the complete execution is observed, and then replaying or discarding the entire sequence to accept or reject it [Ligatti et al. 2003]. The same result holds true of all enforcement mechanisms in our model. If an enforcement mechanism can assume that all executions are finite, it can enumerate them all for any given PM and accept or reject the PM based on the resulting set of possible executions. The interesting computational problems in our model thus hinge upon the fact that complete executions are infinite.

Extending the definition of edit automata to include acceptance or rejection of infinite-length executions is the subject of ongoing research [Ligatti et al. 2005]. We conjecture that because edit automata could be implemented by program-rewriters, they are probably capable of enforcing a subset of the class of RW-enforceable policies.

Shallow History Automata. Fong [2004] investigates the power of execution monitors that are limited by the information that they can recall but that have no restrictions on their computational power. For example, *shallow history automata* can recall the set of events exhibited so far but not the exact order or number of events exhibited. These and other recall-limited EM's are modeled as automata bound by constraints EM1 – EM3 as well as by recollection constraints. On each computational step they observe any security-relevant event about to be exhibited and either (i) accept, allowing the event to be exhibited and continuing the execution, or (ii) reject, preventing the event from being exhibited and terminating execution. As they decide whether to accept or reject, their recollection constraints render them unable to distinguish between certain finite execution prefixes previously observed. Thus they are weaker than the security automata defined in [Schneider 2000], which are constrained only by EM1 – EM3.

For each different recollection constraint imposed on one of these automata, the automaton can enforce a different subclass of the class of policies defined by EM1 – EM3. The set of all possible such constraints gives rise to a lattice of these subclasses [Fong 2004]. At the top of the lattice is the subclass equal to the entire class of policies given by EM1 – EM3, corresponding to the automaton that can distinguish between every pair of execution prefixes. At the bottom of the lattice is the subclass consisting of all policies of the form $\mathcal{P}_B$ for some set B of events as defined in §2.2—that is, policies that prohibit any of a set of "bad" events from being exhibited. Enforcing such a policy does not require the automaton to recall any history of past events that it observed.

The model proposed in [Fong 2004] is incomparable to that presented in this paper because it places no computational constraints on enforcement mechanisms and assumes that all executions are finite. However, if it could be extended to incorporate computational constraints and infinite executions, then this could be used to assess the power of execution monitors that have incomplete access to the event sequences they monitor, such as execution monitors that ignore some of the text of untrusted programs or that cannot observe all non-deterministic choices made by untrusted programs.

Proof–Carrying Code and Certifying Compilers. Proof–Carrying Code (PCC) [Necula 1997; Necula and Lee 1996] and Certifying Compilers [Morrisett et al. 1999; Necula and Lee 1998] are emerging technologies for reducing the trusted computing base needed to enforce security policies. They make it easier for code consumers to enforce security policies by requiring code producers to add proof information to the code that they produce. It is thought easier to write trusted code for verifying proofs than to write trusted code for constructing proofs. PCC is therefore not a single kind of security enforcement mechanism, but rather a framework for reducing and perhaps relocating the trusted computing base. The reader may wonder what policies can be enforced in a PCC framework—that is, the reader may wonder how these technologies fit into the taxonomy of security policies presented in the previous sections. The model and analyses presented in this paper can be used to explore that question, as we now show.

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

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

Alternatively, a theorem prover in a PCC framework might consider the domain of programs to be the set of all object programs. The enforceable policies over this domain are those policies such that for all programs that satisfy the policy, there exists a proof that serves as a witness that the program satisfies the policy. For any proof logic characterizeable by some finite axiomization, this is the set of recursively enumerable (Σ_1) properties of that logic. (If an arbitrary program satisfies the policy, this can be discovered in finite time by enumerating all proofs to find a matching one. But if the program doesn't satisfy the policy, the enumeration process will continue indefinitely without finding a suitable proof.)

In practice, code-proof pairs are usually generated together by some automated

procedure. For example, certifying compilers [Morrisett et al. 1999; Necula and Lee 1998] accept a source program and emit not only object code but also a proof that the object code satisfies some policy. If an arbitrary source program satisfies the policy to be enforced, then the certifying compiler must (i) compile it to object code in a way that faithfully preserves its behavior and (ii) generate a matching proof. If the source program doesn't satisfy the policy, then the compiler must either reject the program (which can be thought of as compiling it to the null program) or compile it to some program that does satisfy the policy, possibly by inserting runtime checks that cause the program to change behaviors when some policy violation would otherwise have occurred. These are precisely conditions RW1 and RW2 from the definition of the RW-enforceable policies. Thus, if one considers the domain of programs to be the set of all source code programs received by a certifying compiler or other automated code-proof pair generator, then the set of enforceable policies are the RW-enforceable policies.

Future Work. The practicality of an enforcement mechanism depends on what resources it consumes. This paper explored the effects of finitely bounding the space and time available to various classes of enforcement mechanisms. However, to be considered practical, real enforcement mechanisms must operate in polynomial or even constant space and time. So an obvious extension to the theory presented here is to investigate (i) the set of policies enforceable by program rewriting when the time and space available to the rewriter is polynomial or constant in the size of the untrusted program and (ii) rewriter functions that produce programs whose size and running time expands by no more than a polynomial or constant in the size and running time of the original untrusted program.

The results of this paper might also be applied to real enforcement mechanisms. SFI [Wahbe et al. 1993], MiSFIT [Small 1997], SASI/PoET/PSLang [Erlingsson and Schneider 2000a; Erlingsson and Schneider 2000b], and Naccio [Evans and Twyman 1999] implement program rewriting but typically assume extremely complex (and mostly unstated) definitions of program equivalence. These equivalence relations would have to be carefully formalized in order to characterize precisely the set of policies that these embodiments of program rewriting actually enforce.

Finally, the class of RW-enforceable policies outside of the coRE policies remains largely unexplored. To investigate this additional power, program rewriting mechanisms must be developed. These would need to accept policy specifications that are not limited to the monitoring-style specifications so easily described by a detector. Consequently, there are interesting questions about how to design a suitably powerful yet usable policy specification language for such a system. For example, various meta-level architectures like Aspect Oriented Programming [Kiczales et al. 1997] have been suggested as general frameworks for enforcing a variety of security policies [Bert Robben and Verbaeten 1999], but it is not clear what class of security policies they can enforce.

6. SUMMARY

Our taxonomy of enforceable security policies is depicted in Figure 2. We have connected this taxonomy to the arithmetic hierarchy of computational complexity theory by observing that the statically enforceable policies are the recursively decid-

able properties and that class EM_{orig} is the coRE properties. We also showed that the RW-enforceable policies are not equivalent to any class of the arithmetic hierarchy. The shaded region in Figure 2 is argued to be a more accurate characterization of the EM-enforceable policies than EM_{orig}.

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

APPENDIX

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

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

— Q is a finite set of *states*.

— δ is the TM's *transition relation*. (Since our TM's are deterministic, δ is a total function.) For each state in Q and each symbol that could be read from the work tape, δ dictates whether the PM halts (H), reads a symbol from the input tape and continues, or continues without reading a symbol from the input tape. If the TM continues without reading an input symbol, then δ specifies the new TM state, the symbol written to the work tape, and whether the work tape head moves left (-1) or right (1). Otherwise if an input symbol is read, it specifies all of the above (the new TM state, the symbol written to the work tape, and whether the work tape header moves left or right) for each possible input symbol seen. Thus δ has type[12]

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

— $q_0 \in Q$ is the initial state of the TM.

— $B \in \Gamma$ is the *blank* symbol to which all cells of the work tape are initialized.

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

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

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

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

according to the following small-step operational semantics:

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

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

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

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

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

where $S \rightarrow_{TM} S'$ and $T : TM \times \Gamma^\omega \times \mathbb{N} \rightarrow \Gamma^*$ is a trace mapping satisfying the constraints on trace mappings given in §2.1. (A concrete example is given below.)

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

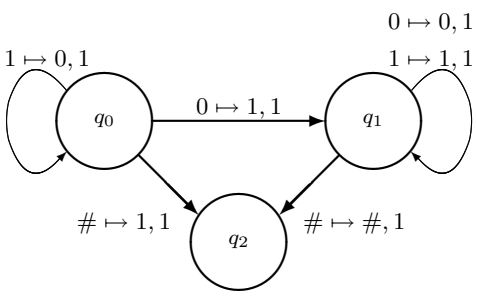
$$E_0 =_{\text{def}} \{e_s | s \in \Gamma_0\} \uplus \{e_{skip}, e_{end}\} \uplus \{e_M | M \in PM\}.$$

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

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

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

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



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

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

where for all $s \in \Gamma_0$,

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

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

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

Fig. 3. A PM for adding one.

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

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

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

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

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

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

Figure 3 shows a program to increment binary numbers by 1, formalized as a PM along the lines we just discussed. The PM shown there treats its input as a two's-complement binary number (least-order bit first), and writes that number incremented by one to its work tape. As the PM executes, it also writes the sequence of symbols dictated by trace mapping T_0 to its trace tape. So if the PM in Figure 3 were provided string 1101 as input, it would write 0011 to its work tape and write $\lceil e_{M_{inc}} \rceil \lceil e_1 \rceil \lceil e_1 \rceil \lceil e_0 \rceil \lceil e_1 \rceil \lceil e_\# \rceil$ to its trace tape, followed by $\lceil e_{end} \rceil$ repeated through the remainder of the tape. A different PM M_0 that never reads its input would write to its trace tape $\lceil e_{M_0} \rceil$, then $\lceil e_{skip} \rceil$ for each computational step it takes, and finally $\lceil e_{end} \rceil$ repeated through the remainder of the tape.

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

ACKNOWLEDGMENTS

The authors wish to thank David Walker for many illuminating discussions about edit automata and other related security automata, and Philip Fong for his comments on shallow history automata. In addition, we wish to thank Tomás Uribe, Úlfar Erlingsson, James Cheney, Matthew Fluet, and Yanling Wang for their helpful comments and critiques.

REFERENCES

Anderson, J. 1972. Computer security technology planning study vols. I and III. Tech. Rep. ESD-TR-73-51, HQ Electronic Systems Division: Hanscom AFB, MA, Fort Washington, Pennsylvania. October.

Bert Robben, Bart Vanhaute, W. J. and Verbaeten, P. 1999. Non-functional policies. *Lecture Notes in Computer Science 1616*, 74–92.

Deutsch, P. and Grant, C. 1971. A flexible measurement tool for software systems. In *Information Processing (Proceedings of the IFIP Congress)*. 320–326.

Erlingsson, Ú. and Schneider, F. B. 2000a. IRM enforcement of Java stack inspection. In *IEEE Symposium on Security and Privacy*. Oakland, California, 246–255.

Erlingsson, Ú. and Schneider, F. B. 2000b. SASI enforcement of security policies: A retrospective. In *WNSP: New Security Paradigms Workshop*. ACM Press.

Evans, D. and Twynman, A. 1999. Flexible policy-directed code safety. In *IEEE Symposium on Security and Privacy*. Oakland, California, 32–45.

Fong, P. W. L. 2004. Access control by tracking shallow execution history. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*. Berkeley, California, 43–55.

Gödel, K. 1931. Über formal unentscheidbare sätze der Principia Mathematica und verwandter Systeme. *Monatshefte für Mathematik und Physik 38*, 173–198.

Harrison, M. A., Ruzzo, W. L., and Ullman, J. D. 1976. Protection in operating systems. *Communications of the ACM 19,* 8 (August), 461–471.

Hopcroft, J. E. and Ullman, J. D. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley.

Kiczales, G., Lamping, J., Medhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. 1997. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, M. Akşit and S. Matsuoka, Eds. Vol. 1241. Springer-Verlag, Berlin, Heidelberg, and New York, 220–242.

Lamport, L. 1977. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering SE-3 2*, 125–143.

Lampson, B. 1971. Protection. In *Proceedings of the 5th Symposium on Information Sciences and Systems*. Princeton, New Jersey, 437–443.

Ligatti, J., Bauer, L., and Walker, D. 2003. Edit automata: Enforcement mechanisms for run-time security policies. Tech. Rep. TR-681-03, Princeton University, Princeton, New Jersey. May.

Ligatti, J., Bauer, L., and Walker, D. 2005. Enforcing non-safety security policies with program monitors. Tech. Rep. TR-720-05, Princeton University, Princeton, New Jersey. January.

Lindholm, T. and Yellin, F. 1999. *The Java™ Virtual Machine Specification*, second ed. Addison-Wesley.

Marcus, L. Fall 1989. The search for a unifying framework for computer security. *IEEE Cipher*, 55–63. Invited Panel Discussion Position Paper, Second Franconia Workshop on Foundations of Computer Security, June 11-14, 1989.

Moonjoo Kim, Sampath Kannan, I. L. and Sokolsky, O. 2001. Java-MaC: a run-time assurance tool for Java. In *1st International Workshop on Run-time Verification*. Paris, France.

Morrisett, G., Crary, K., and Glew, N. 1999. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems 21,* 3 (May), 527–568.

Myers, A. C. 1999. Practical mostly-static information flow control. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages*. San Antonio, Texas, 228–241.

Nachenberg, C. 1997. Computer virus–antivirus coevolution. *Communications of the ACM 40,* 1 (January), 46–51.

Necula, G. 1997. Proof-Carrying Code. In *24th ACM Symposium on Principles of Programming Languages*. Paris, France, 106–119.

Necula, G. C. and Lee, P. 1996. Safe kernel extensions without run-time checking. In *2nd Symposium on Operating Systems Design and Implementation (OSDI '96)*. USENIX, Berkeley, CA, USA, 229–243.

Necula, G. C. and Lee, P. 1998. The design and implementation of a certifying compiler. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 333–344.

Papadimitriou, C. H. 1995. *Computational Complexity*. Addison-Wesley.

Rees, J. and Clinger, W. 1991. Revised 4 report on the algorithmic language Scheme. *Lisp Pointers 4,* 3 (July–September), 1–55.

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

Small, C. 1997. MiSFIT: A tool for constructing safe extensible C++ systems. In *Proceedings of the Third USENIX Conference on Object-Oriented Technologies*. Portland, Oregon.

Turing, A. M. 1936. On computable numbers with an application to the Entscheidungs-problem. In *Proceedings of the London Mathematical Society*. Vol. 2. 42, 230–265.

Viswanathan, M. 2000. Foundations for the run-time analysis of software systems. Ph.D. thesis, University of Pennsylvania.

Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L. 1993. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*. 203–216.

Ware, W. 1979. Security controls for computer systems. Tech. Rep. R-609-1, Rand Corp. October.