

# Computability Classes for Enforcement Mechanisms\*

Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider

Department of Computer Science  
Cornell University  
Ithaca, New York 14853

August 23, 2003

## Abstract

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

## 1 Introduction

Extensible systems, such as web browsers which upload and run applet programs, or operating systems which incorporate drivers for new devices, must ensure that extensions behave in a manner consistent with the intentions of the system designer and its users. When unacceptable behavior goes unchecked, damage can result—not only to the system itself but also to connected systems.

Security enforcement mechanisms are employed to prevent unacceptable behavior. Recently, attention has turned to formally characterizing types of security enforcement mechanisms and identifying the classes of security policies they can enforce [2, 21, 24]. This allows us to assess the power of different security enforcement mechanisms, choose mechanisms well suited to particular security needs, identify which kinds of attacks might still succeed even after a given mechanism has been deployed, and derive meaningful completeness results for newly developed mechanisms.

Schneider [21] suggested an abstract model for security policies and characterized a class of policies meant to capture those that could be effectively enforced through execution monitoring. Viswanathan [24] further refined this characterization by adding a natural computability constraint. We here extend this model to characterize a new class of policies, the *RW-enforceable* policies, corresponding to what can be effectively enforced through program rewriting.

The development and analysis of RW-enforceable policies reveals two subtle flaws in prior work, whereby Schneider’s and Viswanathan’s class admits policies that cannot actually be implemented by any execution monitor. But intersecting their class with the class of RW-enforceable policies

---

\*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.

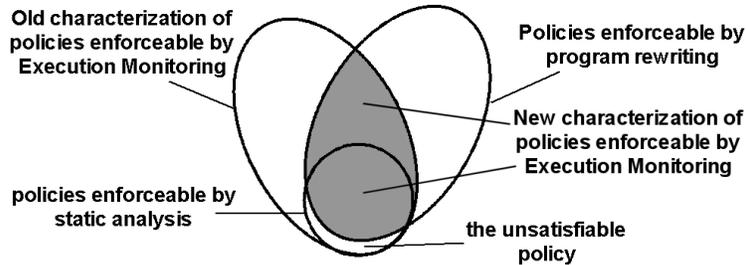


Figure 1: A taxonomy of enforceable security policies.

eliminates these unimplementable policies, yielding a more accurate characterization of what is enforceable by execution monitors, the *EM-enforceable* policies.

Relating the various classes enforceable by execution monitoring and program rewriting reveals the taxonomy of enforceable security policies depicted in Figure 1. Notice that enforcement mechanisms based on program rewriting are able to enforce policies not implementable by any execution monitor. Some of the classes in Figure 1 can be tied to known complexity classes from the arithmetic hierarchy but others cannot.

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 two flaws in prior work on execution monitors. Some related discussion is provided 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 computational systems upon which the untrusted programs themselves are based, because the incompleteness results of Gödel [7] and Turing [23] then imply there will be questions about untrusted programs unanswerable by the enforcement mechanism.

To expose these unanswerable questions, programs must be represented using some model of computation. The Turing Machine (TM) [23] is an obvious candidate because it is well understood; it has already been used to show how computability constraints impinge on the power of access control schemes [8]. Recall that a TM has a finite control comprising a set of states and a transition relation over those states. When a TM moves from one finite control state to another (possibly the same) finite control state in accordance with its transition relation, we will refer to this as a *computational step*. Untrusted programs are modeled herein as a form of TM.

The traditional definition of a TM, as a one-tape finite state machine that accepts or rejects finite-length input strings, does not model non-terminating programs well. Operating systems, which are programs intended to run indefinitely, are expected to read infinitely long strings of input and exhibit behavior not easily characterized in terms of acceptance or rejection of that input. Therefore, untrusted programs will be modeled in this paper by what we term *program machines* (PM)—deterministic TM’s (i.e. TM’s with deterministic transition relations) that manipulate three infinite-length tapes. The three infinite-length tapes manipulated by PM’s are:

- 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. Input tapes may contain any finite or infinite string over some fixed, finite alphabet  $\Gamma$ ; the set of all input strings is denoted by  $\Gamma^\omega$ .
- A *work tape*, which is initially blank and can be read or written by the PM without restriction. It models the work space provided to the program at runtime.
- A write-only *trace tape*, discussed below.

Non-determinism in an untrusted program is modeled by using the input tape contents, even though PM's are themselves deterministic.

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 has written 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 to a PM's trace tape whenever that PM has written a 1 to its work tape. The universe of all observable events  $E$  and their encodings does not vary from PM to PM. We make the following assumptions about  $E$ :

- $E$  is a countably infinite set.
- 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  $M$ .<sup>1</sup> This corresponds to the assumption that the initial memory state of each PM is assumed to be known by the enforcement mechanism.

A means to specify information visible to an enforcement mechanism is essential for realistic modeling of execution monitoring. Typically, some aspects of program execution are visible; others are not. In our model,  $E$  provides the power to express this distinction. 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 reasons why an enforcement mechanism might be unable to enforce a particular security policy. First, 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. Second, 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.

Following [21], program *executions* are modeled as sequences  $\chi$  of events from  $E$ . Complete executions are always infinite event sequences, where program termination is modeled by infinite repetition of a distinguished event  $e_{end}$ . However, since many of our analyses will involve both complete executions and their finite prefixes, we will use the notation  $\chi$  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. Let  $\chi[i]$  (with  $i \geq 1$ ) denote the  $i$ th event of sequence  $\chi$ , let  $\chi[..i]$  denote the length- $i$  prefix of  $\chi$ , and let  $\chi[i+1..]$  denote the suffix of  $\chi$  consisting of all but the first  $i$  events.

---

<sup>1</sup>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.

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. (Since  $E$  is countably infinite, each event can be unambiguously encoded using one or more symbols from finite alphabet  $\Gamma$ .) If the PM terminates, then the encoding of  $e_{end}$  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  $\sigma$ . Let  $X_M$  denote the set of all possible executions exhibited by a PM  $M$  (*viz*  $\{\chi_{M(\sigma)} \mid \sigma \in \Gamma^\omega\}$ ), and let  $X_M^-$  denote the set of all non-empty finite prefixes of  $X_M$  (*viz*  $\{\chi[..i] \mid \chi \in X_M, i \geq 1\}$ ).

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 some 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  $\sigma$ . The trace mapping is left unspecified but assumed to satisfy:

- The first event of every execution exhibited by PM  $M$  is the event  $e_M$ . Thus, 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  $i \geq 1$ .
- Prefix  $\chi_{M(\sigma)}[..i]$  also encodes the prefix of  $\sigma$  read by  $M$  during the first  $i$  steps of its run on input  $\sigma$ . (This is because reading an input symbol is an observable event.) Thus, a security enforcement mechanism can observe the input received during  $M$ ’s run so far.
- No PM performs an infinite number of computational steps without exhibiting any event. If necessary, this constraint can be satisfied by augmenting  $E$  with a special event,  $e_{skip}$ , that indicates that no security-relevant event took place during the given computational step.

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

## 2.2 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, including information flow policies which are defined in terms of the set of all behaviors—and not the individual behaviors in isolation—that a program could possibly exhibit [21]. 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  $e_i \in E$  whenever it writes to cell  $i$  of its work tape, then we might be interested in the security policy  $\mathcal{P}_{boot}$  which 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  $e_i$  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  $\hat{\mathcal{P}}$ .  $\hat{\mathcal{P}}$  will be called a *detector* for  $\mathcal{P}$ . For example, if we define a detector  $\hat{\mathcal{P}}_{boot}(\chi)$  to hold exactly when  $\chi$  does not contain any event  $e_i$  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<sup>2</sup> for an arbitrary execution  $\chi$  by verifying that  $\chi$  does not contain any of a set of *prohibited events*, namely  $e_i$  for  $0 \leq i < 512$ . Such detectors are often useful, so for any set of events  $B \subseteq E$  to be prohibited, we define<sup>3</sup>:

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

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

### 3 Modeling Various Security Enforcement Mechanisms

The framework defined in §2 can be used to model many security enforcement mechanisms, including static analyses [12, 13, 14, 15], reference monitors [1, 11, 12, 20, 24], and program rewriters [3, 4, 5, 6, 22, 25].

#### 3.1 Static Analysis

Enforcement mechanisms that operate strictly prior to running the untrusted program are termed *static analyses*. Here, the enforcement mechanism must accept or reject the untrusted program within a finite period of time.<sup>4</sup> 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<sup>5</sup> [12] and TAL [13]. JFlow [14] and others use static analyses to provide guarantees about other security policies like information flow. Standard virus scanners [15] also implement 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 1.** *The class of statically enforceable security policies is the class of recursively decidable properties of programs (also known as class  $\Pi_0$  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.  $\square$

Class  $\Pi_0$  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  $\Gamma$  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

<sup>2</sup>We say a predicate can be *decided* or is *recursively decidable* iff there exists a terminating algorithm returning 1 for any element that satisfies the predicate and 0 otherwise.

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

<sup>4</sup>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.

<sup>5</sup>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.

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  $\sigma$  as input,” and “ $M$  never terminates.” None of these are recursively decidable for arbitrary PM’s.

### 3.2 Execution Monitoring

Reference monitors [1, 26] and other enforcement mechanisms that operate alongside an untrusted program are termed *execution monitors* (EM’s) in [21]. 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.<sup>6</sup> Examples of EM enforcement mechanisms include access control list and capability-based implementations of access control matrices [11] as well as hardware support for memory protection. Stack inspection performed by the Java Virtual Machine [12], and runtime type-checking such as that employed by dynamically typed languages like Scheme [20], are other examples. The MaC system [24] implements EM’s through a combination of runtime event-checking and program instrumentation.

Schneider [21] observes that for every EM-enforceable security policy  $\mathcal{P}$  there will exist a detector  $\hat{\mathcal{P}}$  such that

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

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

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

and thus the EM-enforceable policies are a subset of the safety properties<sup>7</sup>.

Viswanathan [24] observes that  $\hat{\mathcal{P}}$  must also be computable (something that was left implicit in [21]), giving rise to a fourth restriction:

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

A security policy  $\mathcal{P}$  satisfying EM1 – EM4 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 [10, 21]—satisfies EM1 – EM3 but not EM4.

In §4.1 we show that real EM’s are limited by additional constraints. However, the class described by EM1 – EM4 constitutes a useful upper bound on the set of policies enforceable by execution monitors. Viswanathan [24] shows that EM1 – EM4 is equivalent to the co-recursively enumerable (coRE) properties, also known as class  $\Pi_1$  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. Our model preserves this result, as demonstrated by the proof below.

**Theorem 2.** *The class given by EM1 – EM4 is the class of co-recursively enumerable (coRE) properties of programs (also known as the  $\Pi_1$  class of the arithmetic hierarchy).*

<sup>6</sup>Schneider [21] 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.

<sup>7</sup>A *safety property* is a property that stipulates some “bad thing” does not happen during execution [10].

*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 TM  $M$  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  $\chi$ . We can therefore construct  $M_{\mathcal{P}}$ , as follows: When given  $M$  as input,  $M_{\mathcal{P}}$  begins to iterate through every finite prefix  $\chi$  of executions in  $X_M$ . For each, it decides  $\hat{\mathcal{P}}(\chi)$ . If it finds a  $\chi$  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  $\chi$  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  $M_{\mathcal{P}}(M)$  loops 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, where  $|\chi|$  is the length of sequence  $\chi$ . If  $\chi$  is infinite, then  $\hat{\mathcal{P}}(\chi)$  is true iff  $M_{\mathcal{P}}(\langle\langle\chi\rangle\rangle)$  ever 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  $\chi$  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)  $\chi$ .  $\square$

Since the coRE properties are a proper superset of the recursively decidable properties [19], 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 statically enforceable policies can be enforced by an EM, the static approach is often preferable for engineering reasons. 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. Moreover, this comparison of static enforcement to EM-enforcement assumes that both are being given the same information. If a static analysis is provided one representation of the program (e.g., source code) and an EM is provided another in which some of the information has been erased (e.g., object code), then the static analysis might well be able to enforce policies that the EM cannot.

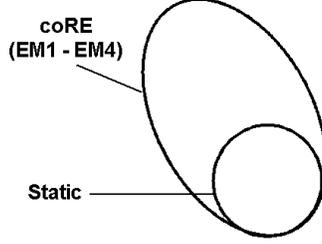


Figure 2: The relationship of the static to the coRE policies.

Theorem 2 also suggests that there are policies enforceable by EM’s that are not statically enforceable, since  $\Pi_0 \subset \Pi_1$ . Policy  $\mathcal{P}_{boot}$  given in §2.2 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 the class characterized by EM1 – EM4 is depicted in Figure 2.

### 3.3 Program Rewriting and $RW_{\approx}$ -Enforceable Policies

*Program rewriting* refers to any enforcement mechanism that, in a finite time, modifies an untrusted program prior to execution. Use of program rewriting for enforcing security policies dates back at least to 1969 [3]. More recently, we find program rewriting being employed in software-based fault-isolation (SFI) [22, 25] as a way of implementing memory-safety policies, in PSLang/PoET [4] and Naccio [6] for enforcing security policies in Java, and in SASI [5] as a means of implementing an *in-lined reference monitor* (IRM) whereby an EM is embedded into the untrusted program. 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 set of program transformations a program rewriter may perform. 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. *PM-equivalence* denoted  $\approx$  of PM’s  $M_1$  and  $M_2$  is defined in terms of an unspecified equivalence relation  $\approx_{\chi}$  on executions.

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

where the requirements on  $\approx_{\chi}$  are:

$$\chi_1 \approx_{\chi} \chi_2 \text{ is recursively decidable}^8 \text{ whenever } \chi_1 \text{ and } \chi_2 \text{ are both finite.} \quad (\text{EQ1})$$

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

EQ1 states that although deciding whether two PM’s are equivalent might be very difficult in general, an enforcement mechanism 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

<sup>8</sup>This assumption can be relaxed to say that  $\chi_1 \approx_{\chi} \chi_2$  is recursively enumerable (RE) without affecting any of our results.

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. EQ2 does not limit program rewriting to this style, however, and EQ2 would be satisfied if, for example, the program rewriting reorders events an untrusted program exhibits. In short, EQ1 and EQ2 admit a broad class of program rewriting whilst ensuring that rewriters preserve certain reasonable inferences about the equivalence of executions and their prefixes.

Given a PM-equivalence relation, we define a policy  $\mathcal{P}$  to be *RW<sub>≈</sub>-enforceable* if there exists a total, computable *rewriter function*  $R : PM \rightarrow PM$  such that for all PM’s  $M$ ,

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

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

Thus, for a security policy  $\mathcal{P}$  to be considered RW<sub>≈</sub>-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 to the old (RW2).

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 limit on which detectors can be considered in our analysis. In particular, it is sensible to consider only consider 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  $\approx_\chi$ . A detector is consistent with  $\approx_\chi$  if 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,  $\approx_\chi$  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 RW<sub>≈</sub>-enforceable policies includes virtually all statically enforceable policies.<sup>9</sup> 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.** *Every satisfiable, statically enforceable policy is RW<sub>≈</sub>-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  $R : TM \rightarrow TM$  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 1, recursively decidable.  $R$  satisfies RW1 because its

---

<sup>9</sup>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.

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

Theorems 2 and 3 together imply that the intersection of the class given by EM1 – EM4 with the  $\text{RW}_{\approx}$ -enforceable policies includes all satisfiable, statically enforceable policies.

The  $\text{RW}_{\approx}$ -enforceable policies that also satisfy EM1 – EM4 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 easily 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 4.** *Assume that PM-equivalence relation  $\approx$  is recursively decidable, and let  $\hat{\mathcal{P}}$  be a detector consistent with  $\approx_{\chi}$ . If the policy  $\mathcal{P}$  induced by  $\hat{\mathcal{P}}$  is  $\text{RW}_{\approx}$ -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 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:

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

Thus,  $\mathcal{P}(M)$  has been decided in finite time and we conclude by Theorem 1 that  $\mathcal{P}$  is statically enforceable.  $\square$

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  $\Pi_2$ -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 program rewriting is not coRE, then program rewriting can be used to enforce policies that are not coRE, and therefore not enforceable by any EM.

**Theorem 5.** *Assume PM-equivalence ( $\approx$ ) is not coRE. There exist policies that are  $RW_{\approx}$ -enforceable but not coRE.*

*Proof.* If  $\approx$  is not coRE, then there is at least one PM  $M_1$  for which deciding  $M \approx M_1$  for arbitrary  $M$  is not coRE. Define  $\mathcal{P}_1(M) =_{\text{def}} (M \approx M_1)$ . Although  $\mathcal{P}_1$  is not coRE, it is  $RW_{\approx}$ -enforceable, because  $R$  can be defined to ignore its input and always yield  $M_1$ . Rewriting function  $R$  satisfies RW1, because  $\mathcal{P}_1(M_1)$  holds. It satisfies RW2, because if  $\mathcal{P}_1(M)$  holds then  $M \approx M_1$  by definition of  $\mathcal{P}_1$ , and hence  $M \approx R(M)$  holds. We conclude that  $\mathcal{P}_1$  is  $RW_{\approx}$ -enforceable.  $\square$

The proof of Theorem 5 gives a simple but practically uninteresting example of an  $RW_{\approx}$ -enforceable policy that is not coRE. Examples of non-coRE,  $RW_{\approx}$ -enforceable policies with real practical import 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.<sup>10</sup> 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, 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 interesting example of such a policy is a kind of least privilege requirement. Suppose programs are *principals* and there is a total, recursively decidable relation  $\sqsubseteq$  over programs that dictates for every pair of programs which has the lesser privilege level on the system.

<sup>10</sup>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.

That is, we write  $M_1 \sqsubset M_2$  if program  $M_1$  has a lesser privilege level than program  $M_2$ . The policy  $\mathcal{P}_{LP}$ , satisfied by exactly those programs  $M$  such that there is no equivalent program  $M' \approx M$  with a lesser privilege level ( $M' \sqsubset M$ ), is not enforceable by any program rewriter.

**Theorem 6.** *Let  $\sqsubset$  be a total, recursively decidable relation over PM's, and assume PM-equivalence ( $\approx$ ) is not recursively decidable. For any PM  $M$  let  $[M]_{\approx}$  denote the set of all programs equivalent to  $M$ , and let  $\min_{\sqsubset}([M]_{\approx})$  denote the minimal PM (with respect to relation  $\sqsubset$ ) in set  $[M]_{\approx}$ . The policy  $\mathcal{P}_{LP}(M) =_{def} (M = \min_{\sqsubset}([M]_{\approx}))$  is not  $RW_{\approx}$ -enforceable.*

*Proof.* Expecting a contradiction, assume that  $\mathcal{P}_{LP}$  is  $RW_{\approx}$ -enforceable. Then there exists a program rewriter  $R$  satisfying RW1 and RW2 that enforces  $\mathcal{P}_{LP}$ . We show that  $R$  can be used to decide PM-equivalence ( $\approx$ ), contradicting the assumption that PM-equivalence is not recursively decidable.

Let arbitrary PM's  $M_1$  and  $M_2$  be given. We can decide whether or not  $M_1 \approx M_2$  by computing  $R(M_1)$  and  $R(M_2)$ . By RW1,  $R(M_1) = \min_{\sqsubset}([M_1]_{\approx})$  and  $R(M_2) = \min_{\sqsubset}([M_2]_{\approx})$ . If  $M_1 \approx M_2$  then  $[M_1]_{\approx} = [M_2]_{\approx}$ , and therefore  $\min_{\sqsubset}([M_1]_{\approx}) = \min_{\sqsubset}([M_2]_{\approx})$ . Hence  $R(M_1) = R(M_2)$ . Alternatively, if  $M_1 \not\approx M_2$  then  $[M_1]_{\approx}$  and  $[M_2]_{\approx}$  are disjoint, and therefore  $\min_{\sqsubset}([M_1]_{\approx}) \neq \min_{\sqsubset}([M_2]_{\approx})$ . Hence  $R(M_1) \neq R(M_2)$ . Thus  $M_1 \approx M_2$  if and only if  $R(M_1) = R(M_2)$ .

Since  $R$  is computable and PM's are of finite size, one can decide whether  $R(M_1) = R(M_2)$  in finite time. This contradicts the assumption that PM-equivalence is not recursively decidable. We conclude that  $\mathcal{P}_{LP}$  is not  $RW_{\approx}$ -enforceable.  $\square$

The ability to enforce policies without explicitly deciding them makes the  $RW_{\approx}$ -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 the class described by EM1 – EM4, no class of the arithmetic hierarchy is equivalent to the class of  $RW_{\approx}$ -enforceable policies.

**Theorem 7.** *Assume PM-equivalence ( $\approx$ ) is not recursively decidable. The class of  $RW_{\approx}$ -enforceable policies is not equivalent to any class of the arithmetic hierarchy.*

*Proof.* Theorem 6 showed that the policy  $\mathcal{P}_{LP}$  is not  $RW_{\approx}$ -enforceable. The proof of Theorem 5 showed that policy  $\mathcal{P}_1$  is  $RW_{\approx}$ -enforceable. Both  $\mathcal{P}_{LP}$  and  $\mathcal{P}_1$  are  $K_i$ -hard, where  $K_i$  is the complexity of the  $\approx$  relation. Since both are  $K_i$ -hard, every class of the arithmetic hierarchy includes or excludes both of them. Since the class of  $RW_{\approx}$ -enforceable policies includes one but not the other, we conclude that it is not equivalent to any class of the arithmetic hierarchy.  $\square$

## 4 Execution Monitors as Program Rewriters

### 4.1 EM-enforceable Policies

The class of security policies described by EM1 – EM4 constitutes an upper bound on the set of EM-enforceable policies, but EM1 – EM4 also admit policies that cannot be enforced by any EM. When an EM detects an impending policy violation, it must intervene and prevent that violation. Such an intervention could be modeled as a finite series of one or more 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  $\mathcal{P}_I$ .<sup>11</sup> Nevertheless,  $\mathcal{P}_I$  satisfies EM1 – EM4 as long as  $I$  is a computable set.

For example, [21] 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 EM1 – EM4. 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 satisfy EM1 – EM4. The power of an EM is thus limited by the set of interventions available to it, in addition to the limitations described by EM1 – EM4.

The power of an EM is also limited by its ability to intervene at an appropriate time in response to a policy violation. Suppose the event  $e_{rel}$  corresponds to releasing a lock for a particular system resource and event  $e_{use}$  corresponds to using that resource, and consider the policies induced by two different detectors, each of which prohibit  $e_{use}$  from occurring after  $e_{rel}$ :

- $\hat{\mathcal{P}}_{RU1}$  stipulates that the EM must intervene at the point that an  $e_{use}$  event is seen after a  $e_{rel}$  has already been seen. (The EM could then enforce the induced policy by preventing the  $e_{use}$  event.) That is,  $\hat{\mathcal{P}}_{RU1}$  is defined by

$$\hat{\mathcal{P}}_{RU1}(\chi) =_{\text{def}} (\forall i : i \geq 1 : (e_{rel} \notin \chi[..i] \vee e_{use} \notin \chi[i+1..])).$$

- $\hat{\mathcal{P}}_{RU2}$  stipulates that the EM must intervene at the point that an  $e_{rel}$  event is seen if some continuation of the current execution might exhibit an  $e_{use}$  event. (The EM could then enforce the induced policy by disallowing the  $e_{rel}$  event in order to preclude an upcoming policy violation.) That is,  $\mathcal{P}_{RU2}$  is defined as the policy induced by detector

$$\hat{\mathcal{P}}_{RU2}(\chi) =_{\text{def}} (e_{rel} \notin \chi \vee (\forall \chi' : \chi\chi' \in X_{\langle\langle\chi\rangle\rangle} : e_{use} \notin \chi')).$$

Detector  $\hat{\mathcal{P}}_{RU2}$  is harder to evaluate than  $\hat{\mathcal{P}}_{RU1}$  because  $\hat{\mathcal{P}}_{RU2}$  requires the EM to simulate the PM arbitrarily far into the future upon observing an  $e_{rel}$  event. This simulation is an uncomputable task, so no real EM could enforce policy  $\mathcal{P}_{RU2}$  in such a way that the EM always intervenes strictly before detector  $\hat{\mathcal{P}}_{RU2}$  is falsified. However, surprisingly, policy  $\mathcal{P}_{RU2}$  satisfies EM1 – EM4. In fact, although detectors  $\hat{\mathcal{P}}_{RU1}$  and  $\hat{\mathcal{P}}_{RU2}$  are different, the policies  $\mathcal{P}_{RU1}$  and  $\mathcal{P}_{RU2}$  that they induce are identical; they agree on all PM's. Any PM that has an execution that violates  $\hat{\mathcal{P}}_{RU1}$  will also have an execution that violates  $\hat{\mathcal{P}}_{RU2}$ . Inversely, a PM whose executions all satisfy  $\hat{\mathcal{P}}_{RU1}$  will also only have executions that satisfy  $\hat{\mathcal{P}}_{RU2}$ .

In conclusion, an EM can “enforce” policy  $\mathcal{P}_{RU1} = \mathcal{P}_{RU2}$  in a way that honors detector  $\hat{\mathcal{P}}_{RU1}$ , but not in a way that honors detector  $\hat{\mathcal{P}}_{RU2}$ . Constraints EM1 – EM4 fail to distinguish between  $\mathcal{P}_{RU1}$  and  $\mathcal{P}_{RU2}$  because they place 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, itself, satisfies the policy, 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

<sup>11</sup>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.

of EM1 – EM4, 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 that satisfy EM1 – EM4 are thus misleading. For example, the MaC system was shown capable of enforcing all coRE policies [24], but policies like  $\mathcal{P}_{RU2}$  cannot be enforced by MaC in such a way as to signal the violations specified by  $\hat{\mathcal{P}}_{RU2}$  before the violation has already occurred.

In §4.2 we will see that the intersection of the class of policies given by EM1 – EM4 with the  $RW_{\approx}$ -enforceable policies constitutes a more suitable characterization of the EM-enforceable policies than the class given by EM1 – EM4 alone. This confirms 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. The intersection of EM1 – EM4 with the  $RW_{\approx}$ -enforceable policies satisfies this intuition by excluding policies like  $\mathcal{P}_{\{e_{end}\}}$  and  $\mathcal{P}_{RU2}$  when interventions that halt the PM (i.e. cause the PM to exhibit  $e_{end}$ ) or that violate detector  $\hat{\mathcal{P}}_{RU2}$  are considered unacceptable. Constraints RW1 and RW2 place demands upon the new set of executions that results when an EM modifies the set of executions that would otherwise be exhibited by a PM. The resulting new set of executions must satisfy the policy to be enforced. This means that it is always possible to take an EM that enforces one of the policies in this intersection, implement it as an in-lined reference monitor, and the result will be a program that satisfies the policy to be enforced.

## 4.2 Benevolent Enforcement of Execution Policies

Detectors for policies that both satisfy EM1 – EM4 and are  $RW_{\approx}$ -enforceable obey a useful property, which we term *benevolence*. A detector  $\hat{\mathcal{P}}$  is defined to be *benevolent* if there exists a decision procedure  $M_{\hat{\mathcal{P}}}$  for finite executions such that for all PM’s  $M$ :

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

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

A policy induced by any detector that satisfies B1 and B2 can be enforced 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  $\hat{\mathcal{P}}_{RU1}$  of §4.1 is an example of a benevolent detector. The decision procedure  $M_{\hat{\mathcal{P}}_{RU1}}(\chi)$  would simply scan  $\chi$  and would reject iff  $e_{use}$  was seen after  $e_{rel}$ . However, detector  $\hat{\mathcal{P}}_{RU2}$  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  $\chi$  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  $M_{\hat{\mathcal{P}}_{RU2}}$  satisfying B1 and B2 exists.

$RW_{\approx}$ -enforceable coRE policies can be always be enforced in a way that prohibits bad events before they occur, yet allows every good program to run. This is because if a coRE policy is  $RW_{\approx}$ -enforceable, then all of its consistent detectors that satisfy EM2 are benevolent.

**Theorem 8.** *Let a detector  $\hat{\mathcal{P}}$  satisfying EM2 be given, and assume that  $\hat{\mathcal{P}}$  is consistent with  $\approx_\chi$ . If the policy  $\mathcal{P}$  induced by  $\hat{\mathcal{P}}$  is  $RW_{\approx}$ -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  $\chi$  as input, it iterates through each  $i \geq 1$  and for each  $i$ , determines if  $\chi \approx_\chi \chi_{R(\langle\langle\chi\rangle\rangle)(\sigma)}[..i]$ , where  $R$  is the rewriter given by the  $RW_{\approx}$ -enforceability of  $\mathcal{P}$  and  $\sigma$  is the string of input symbols recorded in the trace tape as being read during  $\chi$ . Both of these executions are finite, so by EQ1 this can be determined in finite time. If the two executions are equivalent, then  $M_{\hat{\mathcal{P}}}$  halts with acceptance. Otherwise  $M_{\hat{\mathcal{P}}}$  simulates  $M_{\mathcal{P}}(\langle\langle\chi\rangle\rangle)$  for  $i$  steps, where  $M_{\mathcal{P}}$  is a TM that halts if and only if its input represents a PM that violates  $\mathcal{P}$ . Such a TM is guaranteed to exist because  $\mathcal{P}$  satisfies EM1 – EM4 and is therefore coRE by Theorem 2. If  $M_{\mathcal{P}}(\langle\langle\chi\rangle\rangle)$  halts in  $i$  steps, then  $M_{\hat{\mathcal{P}}}$  halts with rejection. Otherwise  $M_{\hat{\mathcal{P}}}$  continues with iteration  $i + 1$ .

First, we prove that  $M_{\hat{\mathcal{P}}}$  always halts. Suppose  $\neg\mathcal{P}(\langle\langle\chi\rangle\rangle)$  holds. Then  $M_{\mathcal{P}}$  will eventually reach a sufficiently large  $i$  that  $M_{\mathcal{P}}(\langle\langle\chi\rangle\rangle)$  will halt, and thus  $M_{\hat{\mathcal{P}}}$  will halt. Suppose instead that  $\mathcal{P}(\langle\langle\chi\rangle\rangle)$  holds. Then by RW2,  $\langle\langle\chi\rangle\rangle \approx R(\langle\langle\chi\rangle\rangle)$ . Applying the definition of  $\approx$ , we see that  $\chi_{\langle\langle\chi\rangle\rangle(\sigma)} \approx_\chi \chi_{R(\langle\langle\chi\rangle\rangle)(\sigma)}$ . Since  $\chi$  is a prefix of  $\chi_{\langle\langle\chi\rangle\rangle(\sigma)}$ , EQ2 implies that there exists a (sufficiently large)  $i$  such that  $\chi \approx_\chi \chi_{R(\langle\langle\chi\rangle\rangle)(\sigma)}[..i]$ . Thus  $M_{\hat{\mathcal{P}}}$  will halt.

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

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

The existence of policies that satisfy EM1 – EM4 but that are not  $RW_{\approx}$ -enforceable can now be shown by demonstrating that there exist coRE policies with detectors that satisfy EM2 but that are not benevolent. By Theorem 8, no such policy can be both coRE and  $RW_{\approx}$ -enforceable.

**Theorem 9.** *There exist detectors  $\hat{\mathcal{P}}$ , that satisfy EM2 and EM3, such that the induced policy defined by  $\mathcal{P}(M) =_{\text{def}} (\forall\chi : \chi \in X_M : \hat{\mathcal{P}}(\chi))$  satisfies EM1 – EM4, and yet  $\hat{\mathcal{P}}$  is not benevolent.*

*Proof.* Define  $\mathcal{P}_{\{e_{\text{end}}\}}$  as in §4.1 and define  $\hat{\mathcal{P}}_{NT}(\chi) =_{\text{def}} \mathcal{P}_{\{e_{\text{end}}\}}(\langle\langle\chi\rangle\rangle)$ . That is, an execution satisfies  $\hat{\mathcal{P}}_{NT}$  if and only if it comes from a program that never halts on any input. Observe that  $\hat{\mathcal{P}}_{NT}$  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  $\mathcal{P}_{NT} \equiv \mathcal{P}_{\{e_{\text{end}}\}}$  by the following line of reasoning:

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

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

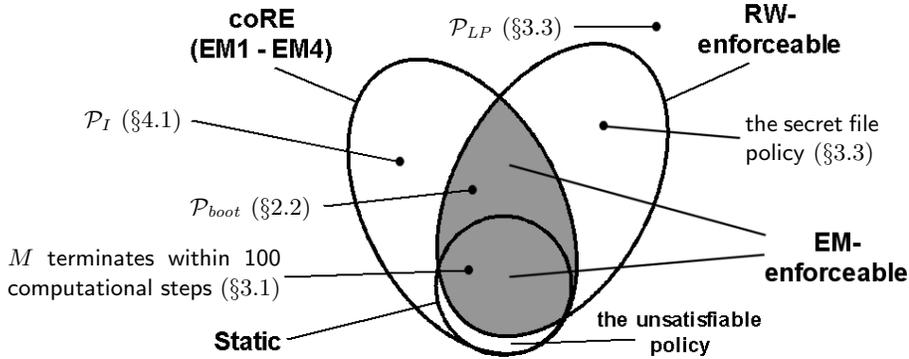


Figure 3: Classes of security policies and some policies that lie within them.

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  $M$ , compute  $M_{\hat{\mathcal{P}}_{NT}}(\chi_{M(\sigma)}[..1])$ , where  $M_{\hat{\mathcal{P}}_{NT}}$  is the decision procedure predicted to exist by the benevolence of  $\hat{\mathcal{P}}_{NT}$ , and  $\sigma$  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.  $\square$

The relationship of the statically enforceable policies, the class given by EM1 – EM4 (the coRE policies), and the policies enforceable by program rewriting (the  $RW_{\approx}$ -enforceable policies) is summarized in Figure 3. The statically enforceable policies are a subset of the coRE policies and, with the exception of the unsatisfiable policy, a subset of the  $RW_{\approx}$ -enforceable policies. The shaded region indicates those policies that are both coRE and  $RW_{\approx}$ -enforceable. These are the policies that we characterize as EM-enforceable. There exist coRE policies outside this intersection, but all such policies are induced by some non-benevolent detector. Thus, using an EM to “enforce” any of these policies would result in program behavior that might continue to exhibit events that the policy was intended to prohibit. There are also  $RW_{\approx}$ -enforceable policies outside this intersection. These are policies that cannot be enforced by an EM but that can be enforced by a program rewriter that does not limit its rewriting to producing in-lined reference monitors.

Additionally, Figure 3 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 EM1 – EM4 but that cannot be enforced by any real EM in a way that prevents bad events from occurring on the system.

### 4.3 Edit Automata

*Edit automata* [2] 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. If an edit automaton intervenes during execution, then it is deemed to have rejected that execution; if not, it is deemed to have accepted the execution.

The operational semantics of edit automata [2] permit any edit automaton to be simulated by a TM. The TM accepts an encoding of an execution on its input tape, simulates the computational steps that the edit automaton would take, and writes an encoding of a new execution to its work tape. Using such a simulation, it can be shown that any security policy enforceable by an edit automaton satisfies EM1 – EM4.

**Theorem 10.** *All policies enforceable by edit automata satisfy EM1 – EM4.*

*Proof.* Let a security policy  $\mathcal{P}$  be given and assume  $\mathcal{P}$  is enforceable by an edit automaton  $A$ . There exists a TM  $M_A$  that accepts an encoding of an arbitrary execution  $\chi$  as input and writes an encoding of  $A(\chi)$ , the behavior of edit automaton  $A$  on  $\chi$ , to its work tape. Edit automata [2] have a small-step operational semantics that include four types of steps:

- **E-StepA:** The edit automaton permits the untrusted program to exhibit an event.
- **E-StepS:** The automaton suppresses an event that the untrusted program would otherwise have exhibited.
- **E-Ins:** The automaton inserts a sequence of events in place of an event that the untrusted program would otherwise have exhibited.
- **E-Stop:** The automaton prematurely terminates the untrusted program in place of the event it would otherwise have exhibited.

The definition given in [2] of execution acceptance for edit automata dictates that  $A$  rejects  $\chi$  if and only if it performs operation **E-StepS**, **E-Ins**, or **E-Stop** at any point when provided  $\chi$  as input. Define a new TM  $M'_A$ , exactly like  $M_A$  except with a transition relation that causes the TM to halt in lieu of performing simulations of any of these three edit automaton operations. Additionally, define the transition relation of  $M'_A$  such that  $M'_A$  will loop if it ever reaches the end of its input without halting. TM  $M'_A(\chi)$  will therefore halt if and only if  $A$  would accept  $\chi$  and loop otherwise. The existence of TM  $M'_A$  satisfies the definition of coRE. Theorem 2 establishes that all coRE policies satisfy EM1 – EM4. Therefore we conclude that  $\mathcal{P}$  satisfies EM1 – EM4.  $\square$

Although edit automata were not defined in terms of program rewriting, the cumulative effect of an edit automaton can be regarded as a restricted form of program rewriting. Given an arbitrary edit automaton, define  $r : E^\omega \rightarrow E^\omega$  to be a function that maps an arbitrary execution  $\chi$  to the execution that results from the edit automaton's action on  $\chi$ . Then define rewriter function  $R_r$  to map  $M$  an arbitrary PM to a new PM  $M_r$  such that  $X_{M_r} = \{e_{M_r}, r(\chi_{M(\sigma)}) \mid \sigma \in \Gamma^\omega\}$ .  $M_r$  is  $M$  modified on each step by the edit automaton described by  $r$ .<sup>12</sup> Rewriter function  $R_r$  can be constructed provided the act of computing  $r$  does not itself produce inequivalent executions. That is, letting events exhibited during the course of computing  $r$  be called  $r$ -events, two executions are defined to be equivalent ( $\approx_\chi^{EA}$ ) if the executions are identical after all  $r$ -events and the initial events ( $e_M$ ) are removed from both. Now, if a policy is enforceable by an edit automaton, then it is  $RW_{\approx^{EA}}$ -enforceable. And since  $\approx_\chi^{EA}$  satisfies EQ1 and EQ2, given that edit automata enforce only policies that satisfy EM1 – EM4, Theorem 8 establishes that all policies enforced by Edit Automata can be enforced with benevolent detectors. In conclusion, the set of policies enforceable by edit automata is a subset of the intersection of the coRE policies with the  $RW_{\approx^{EA}}$ -enforceable policies.

<sup>12</sup>Recall from §2.1 that  $e_{M_r}$  is the event that encodes the initial memory state of PM  $M_r$ .

## 5 Discussion

**Program-agnostic Security Enforcement.** The existence of a function  $\langle\langle\cdot\rangle\rangle$  that maps executions to the PM's that generated them is crucial for some of the results in this paper. The assumption that  $\langle\langle\cdot\rangle\rangle$  exists is realistic, in so far as enforcement mechanisms located in the processor or operating system have access to the program. However, only superficial use has been made of this information in actual EM implementations to date.

If the availability of a function  $\langle\langle\cdot\rangle\rangle$  is no longer assumed, then our taxonomy of policy classes changes slightly to exclude from the EM-enforceable policies those policies that depend on information obtained using  $\langle\langle\cdot\rangle\rangle$ . For example, statically enforceable policies are no longer a subset of the EM-enforceable policies, because a static mechanism can distinguish between PM's whose sets of executions are identical while an EM cannot. However, within the domain of policies that depend solely on information that is available both to a static analysis and an EM, the taxonomy is unchanged.

In a world without  $\langle\langle\cdot\rangle\rangle$ , details of the computational model become more important. An EM, for instance, must be able to permit or disallow newly exhibited events before they affect the system. Our analyses did not need to assume anything about how far into the future an EM could look to predict an event, because our EM's could simulate  $\langle\langle\chi\rangle\rangle$  on various inputs to look finitely far into the possible futures of  $\chi$ . However, if  $\langle\langle\cdot\rangle\rangle$  does not exist, one must make specific assumptions about which future events an EM can predict and under what conditions it can do so. Such details about the computational system were conveniently abstracted by our model.

**Proof-Carrying Code and Certifying Compilers.** In Proof-Carrying Code [16, 17], mobile code transmitted to an untrusting recipient is paired with a proof that the code satisfies whatever safety policy is being demanded by the untrusting recipient. The untrusting recipient can check 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 untrusting recipient.

The class of policies enforceable by Proof-Carrying Code depends on what is the domain of all programs. If the domain of programs is taken to be the set of all object code-proof pairs receiveable by the untrusting recipient, then the set of enforceable security policies 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, one can take the domain of programs to be the set of all object programs receiveable by the untrusting recipient. 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 ( $\Sigma_1$ ) 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.)

The above analyses make no assumption about the origin of proofs or programs; programs and proofs might be generated by an oracle of unlimited computational power. In practice, however, code-proof pairs are usually generated by some automated procedure. For example, certifying compilers [13, 18] 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 will depend 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 available time and space is polynomial 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 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 [25], MiSFIT [22], SASI/PoET/PSLang [4, 5], and Naccio [6] 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. In addition, it is not known whether the power of generalized frameworks for program rewriting, like Aspect Oriented Programming [9], can be characterized in similar ways.

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.

## 6 Summary

Our taxonomy of enforceable security policies is depicted in Figure 3. We have connected this taxonomy to the arithmetic hierarchy of computational complexity theory by observing that the statically enforceable policies are the recursively decidable properties and that the class given by one characterization of the *EM*-enforceable policies (given by EM1 – EM4) are 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 3 is argued to be a more accurate characterization of the *EM*-enforceable policies than EM1 – EM4.

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.

## Acknowledgements

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

## References

- [1] J.P. Anderson. Computer security technology planning study vols. i and iii. Technical Report ESD-TR-73-51, HQ Electronic Systems Division: Hanscom AFB, MA, Fort Washington, Pennsylvania, October 1972.
- [2] Lujo Bauer, Jarred Ligatti, and David Walker. More enforceable security policies. Technical Report TR-649-02, Princeton University, Princeton, New Jersey, June 2002.
- [3] P. Deutsch and C.A. Grant. A flexible measurement tool for software systems. In *Information Processing (Proceedings of the IFIP Congress)*, pages 320–326, 1971.
- [4] Úlfar Erlingsson and Fred B. Schneider. IRM enforcement of Java stack inspection. In *IEEE Symposium on Security and Privacy*, pages 246–255, Oakland, California, May 2000.
- [5] Úlfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: A retrospective. In *WNSP: New Security Paradigms Workshop*. ACM Press, 2000.
- [6] David Evans and Andrew Twynman. Flexible policy-directed code safety. In *IEEE Symposium on Security and Privacy*, pages 32–45, Oakland, California, May 1999.
- [7] Kurt Gödel. Über formal unentscheidbare sätze der Principia Mathematica und verwandter Systeme. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.
- [8] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, August 1976.
- [9] Gregor Kiczales, John Lamping, Anurag Medhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.
- [10] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering SE-3*, 2:125–143, March 1977.
- [11] B. Lampson. Protection. In *Proceedings of the 5th Symposium on Information Sciences and Systems*, pages 437–443, Princeton, New Jersey, March 1971.
- [12] Tim Lindholm and Frank Yellin. *The Java™ Virtual Machine Specification*. Addison-Wesley, second edition, 1999.
- [13] Greg Morrisett, Karl Cray, and Neal Glew. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.
- [14] Andrew C. Myers. Practical mostly-static information flow control. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages*, pages 228–241, San Antonio, Texas, January 1999.

- [15] Carey Nachenberg. Computer virus–antivirus coevolution. *Communications of the ACM*, 40(1):46–51, January 1997.
- [16] George Necula. Proof-Carrying Code. In *24th ACM Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, January 1997.
- [17] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *2nd Symposium on Operating Systems Design and Implementation (OSDI '96)*, pages 229–243, Berkeley, CA, USA, 1996. USENIX.
- [18] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 333–344, 1998.
- [19] Christos H. Papadimitriou. *Computational Complexity*, page 63. Addison-Wesley, 1995.
- [20] J. Rees and W. Clinger. Revised 4 report on the algorithmic language Scheme. *Lisp Pointers*, 4(3):1–55, July–September 1991.
- [21] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and Systems Security*, 3(1):30–50, February 2000.
- [22] Christopher Small. MiSFIT: A tool for constructing safe extensible C++ systems. In *Proceedings of the Third USENIX Conference on Object-Oriented Technologies*, Portland, Oregon, June 1997.
- [23] A. M. Turing. On computable numbers with an application to the Entscheidungs-problem. In *Proceedings of the London Mathematical Society*, volume 2, pages 42, 230–265, 1936.
- [24] Mahesh Viswanathan. *Foundations for the Run-time Analysis of Software Systems*. PhD thesis, University of Pennsylvania, December 2000.
- [25] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 203–216, December 1993.
- [26] W. Ware. Security controls for computer systems. Technical Report R-609-1, Rand Corp, October 1979.

## A Formal PM Construction

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

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

- $Q$  is a finite set of *states*.
- $\delta$  is the TM’s *transition relation*. For each state in  $Q$  and each symbol that could be read from the work tape,  $\delta$  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  $\delta$  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  $\delta$  has type<sup>13</sup>

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

- $q_0 \in Q$  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. Initially, TM  $M = (Q, \delta, q_0, B)$  when provided input  $\sigma$  begins in computation state  $\langle q_0, \sigma, 1, B^\omega, 1 \rangle$ . The TM computation state then changes according to the following small-step operational semantics:

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

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:

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

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  $\sigma$  begins in computation state  $\langle S, \cdot, 0 \rangle$  where  $S$  is the initial computation state of TM  $M$  for input  $\sigma$ . 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 will be given shortly.)

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  $\Gamma_0$  be  $\{0, 1, \#\}$ . Next define event set  $E_0$  by

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

$E_0$  is a countable set, so there exists an unambiguous encoding of events from  $E_0$  as finite sequences of symbols from  $\Gamma_0$ . Choose such an encoding and let  $[e]$  denote the encoding of event  $e \in E_0$ . To

<sup>13</sup>Set operator  $\uplus$  denotes disjoint union.

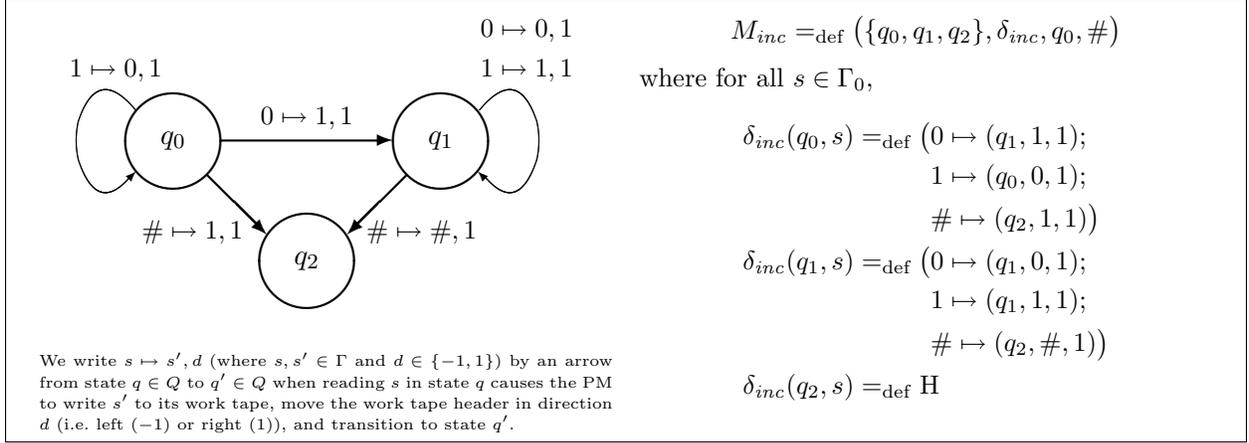


Figure 4: A PM for adding one.

avoid ambiguities in representing event sequences, choose the encoding so that for all  $e \in E_0$ , string  $[e]$  consists only of symbols in  $\{0, 1\}$  followed by a  $\#$ . This ensures that there exists a computable function  $[\cdot] : \Gamma^\omega \rightarrow E^\omega$  such that for all  $i \geq 0$  and for all  $\chi \in E^i$ ,  $[[e_0] \cdots [e_i]] = e_0 \dots e_i$ .

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

$$T_0(M, \sigma, 0) =_{\text{def}} [e_M].$$

$$T_0((Q, q_0, \delta, B), \sigma, n + 1) =_{\text{def}} [e_{\sigma[i]}] \quad \text{if } \langle q_0, \sigma, 1, B^\omega, 1 \rangle \xrightarrow{n}_{TM} \langle q, \sigma, i, \kappa, k \rangle, \text{ and}$$

$$\delta(q, \kappa[k]) \in (\Gamma \rightarrow (Q \times \Gamma \times \{-1, 1\})).$$

$$T_0((Q, q_0, \delta, B), \sigma, n + 1) =_{\text{def}} [e_{end}] \quad \text{if } \langle q_0, \sigma, 1, B^\omega, 1 \rangle \xrightarrow{n}_{TM} \langle q, \sigma, i, \kappa, k \rangle, \text{ and}$$

$$\delta(q, \kappa[k]) = H.$$

$$T_0(M, \sigma, n + 1) =_{\text{def}} [e_{skip}] \quad \text{otherwise.}$$

So, this trace mapping causes every PM  $M$  to write  $[e_M]$  to its trace tape before its first computational step, write  $[e_s]$  whenever it reads symbol  $s$  from its input tape, write  $[e_{skip}]$  whenever it does not read an input symbol on a given computational step, and pad the remainder of the trace tape with  $[e_{end}]$  if it halts.

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

$$\chi_{M(\sigma)} =_{\text{def}} [\tau]$$

where  $\tau$  is the limit as  $n \rightarrow \infty$  of

$$\langle \langle q_0, \sigma, 1, B^\omega, 1 \rangle, \cdot, 0 \rangle \xrightarrow{n}_{TM} \langle \langle q, \sigma, i, \kappa, k \rangle, \tau, n \rangle$$

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 4 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 4 were provided string 1101 as input, it would write 0011 to its work tape and write  $[e_{M_{inc}}][e_1][e_1][e_0][e_1][e_\#]$  to its trace tape, followed by  $[e_{end}]$  repeated through the

remainder of the tape. A different PM  $M_0$  that never reads its input would write to its trace tape  $[e_{M_0}]$ , then  $[e_{skip}]$  for each computational step it takes, and finally  $[e_{end}]$  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. TM's can also simulate other TM's, which means they can perform the equivalent of runtime code generation. Turing Machines are thus an extremely flexible model of computation that can be used to simulate real computer architectures.