Run-time Enforcement of Nonsafety Policies

JAY LIGATTI
University of South Florida
LUJO BAUER
Carnegie Mellon University
and
DAVID WALKER
Princeton University

A common mechanism for ensuring that software behaves securely is to monitor programs at run time and check that they dynamically adhere to constraints specified by a security policy. Whenever a program monitor detects that untrusted software is attempting to execute a dangerous action, it takes remedial steps to ensure that only safe code actually gets executed.

This article improves our understanding of the space of policies enforceable by monitoring the run-time behaviors of programs. We begin by building a formal framework for analyzing policy enforcement: we precisely define policies, monitors, and enforcement. This framework allows us to prove that monitors enforce an interesting set of policies that we call the infinite renewal properties. We show how, when given any reasonable infinite renewal property, to construct a program monitor that provably enforces that policy. We also show that the set of infinite renewal properties includes some nonsafety policies, i.e., that monitors can enforce some nonsafety (including some purely liveness) policies. Finally, we demonstrate concrete examples of nonsafety policies enforceable by practical run-time monitors.

Categories and Subject Descriptors: D.2.0 [**Software Engineering**]: General—*protection mechanisms*; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*interactive and reactive computation*; D.2.5 [**Software Engineering**]: Testing and Debugging—*monitors*; D.2.4 [**Software Engineering**]: Software/Program Verification—*validation*; *formal methods*

General Terms: Security, Theory, Experimentation

Additional Key Words and Phrases: Security policies, safety, liveness, monitoring, security automata, policy enforcement

1. INTRODUCTION

A ubiquitous technique for enforcing software security is to dynamically monitor the behavior of programs and take remedial action when the programs behave in ways that violate a security policy. Firewalls, virtual machines, anti-virus and anti-spyware programs, intrusion-detection tools, and operating systems all act as *program monitors* to enforce security policies in this way. We can also think of any application containing security code that dynamically checks input values, queries network configurations, raises exceptions, warns the user of potential consequences of opening a file, etc., as containing a program monitor *inlined* into the application. Even "static" mechanisms, such as type-safe-language compilers and verifiers, often ensure that programs contain appropriate dynamic checks by inlining them into the code. This article examines the space of policies enforceable by program monitors.

Because program monitors, which react to the potential security violations of *target programs*, enjoy such ubiquity, it is important to understand their capabilities as policy enforcers. Such an understanding is essential for developing sound systems that support program monitoring and languages for specifying the security policies that these systems can enforce. In addition, well-defined boundaries on the enforcement powers of security mechanisms allow security architects to determine exactly when certain mechanisms are needed and save the architects from attempting to enforce policies with insufficiently strong mechanisms.

Schneider defined the first formal models of program monitors and discovered one particularly useful boundary on their power [Schneider 2000]. He defined a class of monitors that respond to potential security violations by halting the target application, and he showed that these monitors can only enforce *safety* properties—security policies that specify that "nothing bad ever happens" in a valid run of the target [Lamport 1977]. When a monitor in this class detects a potential security violation (i.e., "something bad"), it must halt the target.

Aside from our work, other research has likewise only focused on the ability of program monitors to enforce safety properties. In this article, we advance our theoretical understanding of practical program monitors by proving that certain types of monitors can enforce nonsafety properties. These monitors are modeled by *edit automata*, which have the power to insert actions on behalf of, and suppress actions attempted by, the target application. We prove an interesting lower bound on the properties enforceable by such monitors—a lower bound that encompasses strictly more than safety properties. We also detail several nonsafety policies that we have enforced in practice using an implemented monitoring enforcement system called Polymer [Bauer et al. 2005a; Ligatti 2006].

1.1 Related Work

Only a handful of efforts have been made to understand the space of policies enforceable by monitoring software at run time. In contrast, a rich variety of monitoring enforcement systems has been implemented [Liao and Cohen 1992; Jeffery et al. 1998; Edjlali et al. 1998; Damianou et al. 2001; Erlingsson and Schneider 2000; 1999; Evans and Twyman 1999; Evans 2000; Robinson 2002; Kim et al. 1999; Bauer et al. 2003; Erlingsson 2003; Sen et al. 2004; Havelund and Roşu 2004]. This lack of theoretical work makes it difficult to understand exactly which sorts of security policies the implemented systems can enforce. In this section we examine closely related efforts and discuss high-level similarities and differences between them and our work. In the remainder of this article, we point out additional, more specific relationships between our results and those of related work.

Monitors as Invalid Execution Recognizers. Schneider began the effort to understand the space of policies that monitors can enforce [Schneider 2000]. Building on earlier work with Alpern, which provided logic-based and automata-theoretic definitions of safety and liveness [Alpern and Schneider 1985; 1987], Schneider modeled program monitors as infinite-state automata using a particular variety of Büchi automata [Büchi 1962] (which are like regular deterministic finite automata except that they can have an infinite number of states, operate on infinite-length input strings, and accept inputs that cause the automaton to enter accepting states infinitely often). Schneider's monitors[1] observe executions of untrusted target applications and dynamically recognize invalid behaviors. When a monitor recognizes an invalid execution, it halts the target just before the execution becomes invalid, thereby guaranteeing the validity of all monitored executions. Schneider formally defined policies and properties and observed that his automata-based execution recognizers can only enforce safety properties (a monitor can only halt the target upon observing an irremediably "bad" action).

This article builds on Schneider's definitions and models but views program monitors as execution *transformers* rather than execution *recognizers*. This fundamental shift permits modeling the realistic possibility that a monitor might *insert* actions on behalf of, and *suppress* actions of, untrusted target applications. In our model, Schneider's monitors are *truncation automata*, which can only *accept* the actions of untrusted targets and *halt* the target altogether upon recognizing a safety violation. We define more general monitors modeled by *edit automata* that can insert and suppress actions (and are therefore operationally similar to deterministic I/O automata [Lynch and Tuttle 1987]), and we prove that edit automata are strictly more powerful than truncation automata (Section 3.2.2). We demonstrate concrete monitors that enforce nonsafety properties, and even pure liveness properties, in Section 5.

Computability Constraints on Execution Recognizers. After Schneider showed that the safety properties constitute an upper bound on the set of policies enforceable by simple monitors, Viswanathan, Kim, and others tightened this bound by placing explicit computability constraints on the safety properties being enforced [Viswanathan 2000; Kim et al. 2002]. Their key insight was that because execution recognizers inherently have to decide whether target executions are invalid, these monitors can only enforce decidable safety properties. Introducing computability constraints allowed them to show that monitors based on recognizing invalid executions (i.e., our truncation automata) enforce exactly the set of computable safety properties. Moreover, Viswanathan proved that the set of languages containing strings that satisfy a computable safety property equals the set of coRE languages [Viswanathan 2000].

Shallow-history Execution Recognizers. Continuing the analysis of monitors acting as execution recognizers, Fong defines *shallow history automata* (SHA) as a specific type of memory-bounded monitor [Fong 2004]. SHA decide whether to accept an action by examining a finite and unordered history of previously accepted actions. Although SHA are very limited models of finite-state truncation automata, Fong shows that they can nonetheless enforce a wide range of useful access-control

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

properties, including Chinese Wall policies (where subjects may access at most one element from every set of conflicting data [Brewer and Nash 1989]), low-water-mark policies (where a lattice of trustworthiness determines whether accesses are valid [Biba 1975]), and one-out-of-k authorization policies (where every program has a predetermined, finite set of access permissions [Edjlali et al. 1998]). In addition, Fong generalizes SHA by defining sets of properties accepted by arbitrarily memory-bounded monitors and proves that classes of monitors with strictly more memory can enforce strictly more properties.

Fong simplifies his analyses by assuming that monitors observe only finite executions (i.e., all untrusted targets must eventually halt) and ignoring computability constraints on monitors. Although we do not make those simplifying assumptions in this article, we did when first exploring the capabilities of edit automata [Bauer et al. 2002; Ligatti et al. 2005a].

Comparison of Enforcement Mechanisms' Capabilities. Hamlen, Morrisett, and Schneider observe that, in practice, program monitors are often implemented by *rewriting* untrusted target code [Hamlen et al. 2006]. A rewriter *inlines* a monitor's code directly into the target at compile or load time. Many of the implemented monitoring systems cited at the beginning of this subsection can be viewed as program rewriters.

Hamlen et al. define the set of *RW-enforceable policies* as the policies enforceable by rewriting untrusted target applications, and they compare this set with the sets of policies enforceable by static analysis and monitoring mechanisms. Their model of program monitors differs from ours in that their monitors have access to the full text (e.g., source code or binaries) of monitored target programs. Practical monitors often adhere to this assumption: operating systems and virtual machines can usually access the full code of target programs. However, practical monitors also often violate this assumption: firewalls, network scanners, and monitors that can only hook their code into security-relevant methods of an operating system API (such as the "cloaking" monitors installed by some DRM mechanisms [Russinovich 2005]) lack access to target programs' code.

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

1.2 Contributions

We extend previous work in four primary ways.

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

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

(3) We analyze the set of infinite renewal properties to determine its relationships with the standard sets of safety and liveness policies (Section 4). We prove that the set of infinite renewal properties includes some nonsafety properties and, hence, that program monitors are in theory capable of enforcing some nonsafety properties.

(4) We describe concrete examples of monitors that we have implemented to enforce nonsafety properties (Section 5). The existence of these concrete examples validates our theory and demonstrates some strategies for implementing nonsafety-property enforcers.

Large portions of this article describing theoretical definitions of policies, monitors, and enforcement, as well as the properties enforceable by program monitors, come from a recent paper presented at the Tenth European Symposium on Research in Computer Security (ESORICS) in September 2005 and entitled "Enforcing Nonsafety Security Policies with Program Monitors" [Ligatti et al. 2005b]. This article extends that paper in the following ways.

—We have given related work a much more complete treatment in Section 1.1.

—We include proofs for the theorems in Section 3. These theorems delineate upper and lower bounds for the properties enforceable by various monitoring mechanisms. Most importantly, the proof of Theorem 3.4 shows how, when given any reasonable infinite renewal property, to construct a program monitor that provably enforces that property.

—We include, in the all-new Section 5, examples of nonsafety policies demonstrably enforceable in practice. We have enforced these nonsafety policies in an implemented system called Polymer [Bauer et al. 2005a; Ligatti 2006], the source code for which is publicly available [Bauer et al. 2005b].

—Finally, we make numerous minor corrections and additions to the original material. For instance, Section 2.1 includes a correction in our notation for sequence concatenation (the original notation only applied when concatenating two finite sequences, but we often need this notation to denote a concatenation comprised of a finite sequence followed by an infinite sequence).

2. MODELING MONITORS AS SECURITY AUTOMATA

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

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

2.1 Notation

We specify a system at a high level of abstraction as a nonempty, possibly countably infinite set of *program actions* $\mathcal{A}$ (also referred to as program events). An *execution* is simply a finite or infinite sequence of actions. The set of all finite executions on a system with action set $\mathcal{A}$ is notated as $\mathcal{A}^\star$. Similarly, the set of infinite executions is $\mathcal{A}^\omega$, and the set of all executions (finite and infinite) is $\mathcal{A}^\infty$. We let the metavariable a range over actions, σ and τ over executions, and Σ over sets of executions (i.e., subsets of $\mathcal{A}^\infty$).

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

2.2 Policies and Properties

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

Following Schneider [Schneider 2000], we distinguish between *properties* and more general policies as follows. A security policy P is a *property* if and only if there exists a *characteristic predicate* $\hat{P}$ over $\mathcal{A}^\infty$ such that for all $\Sigma \subseteq \mathcal{A}^\infty$, the following is true.

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

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

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

Properties that specify that "nothing bad ever happens" are called *safety properties* [Lamport 1977; Alpern and Schneider 1985]. No prefix of a valid execution can violate a safety property; equivalently, once some finite execution violates the property, all extensions of that execution violate the property. Technically, safety means that every invalid execution has some invalid prefix after which all extensions are likewise invalid. Formally, $\hat{P}$ is a safety property on a system with action set $\mathcal{A}$ if and only if the following is true.[2]

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

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

Dually to safety properties, *liveness properties* [Alpern and Schneider 1985] state that nothing irremediably bad happens in any finite amount of time. Any finite sequence of actions can always be extended so that it satisfies the property. Formally, $\hat{P}$ is a liveness property on a system with action set $\mathcal{A}$ if and only if the following is true.

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

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

General properties may allow executions to alternate freely between satisfying and violating the property. Such properties are neither safety nor liveness but instead a combination of a single safety and a single liveness property [Alpern and Schneider 1987]. We show in Sections 3 and 4 that edit automata effectively enforce an interesting new sort of property that is neither safety nor liveness.

2.3 Security Automata

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

[2]Alpern and Schneider [Alpern and Schneider 1985] model executions as infinite-length sequences of states in which terminating executions contain a final state, infinitely repeated. We can map an execution in their model to one in ours simply by sequencing the events that induce the state transitions (no event induces a repeated final state). With this mapping, it is easy to verify that our definitions of safety and liveness are equivalent to those of Alpern and Schneider.

definition of each kind of security automaton; we only require that δ be complete, deterministic, and Turing Machine computable. We limit our analysis in this work to automata whose transition functions take the current state and input action (the next action the target wants to execute) and return a new state and at most one action to output (make observable). The current input action may or may not be consumed while making a transition.

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

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

Definition 2.1 *Multi-step.* The multi-step relation $(\sigma, q) \overset{\tau}{\Longrightarrow}_S (\sigma', q')$ is inductively defined as follows (where all metavariables are universally quantified).

(1) $(q, \sigma) \overset{\cdot}{\Longrightarrow}_S (q, \sigma)$

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

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

Definition 2.2 *Transforms.* A security automaton $S = (Q, q_0, \delta)$ on a system with action set $\mathcal{A}$ *transforms* the input sequence $\sigma \in \mathcal{A}^\infty$ into the output sequence $\tau \in \mathcal{A}^\infty$, notated as $(q_0, \sigma) \Downarrow_S \tau$, if and only if the following two constraints are met.

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

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

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

2.4 Property Enforcement

We and several other authors have concurrently noted the importance of monitors obeying two abstract principles, which we call *soundness* and *transparency* [Ligatti et al. 2003; Hamlen et al. 2006; Erlingsson 2003]. A mechanism that purports to enforce a property $\hat{P}$ is *sound* when it ensures that observable outputs always obey $\hat{P}$; it is *transparent* when it preserves the semantics of executions that already obey $\hat{P}$. We call a sound and transparent mechanism an *effective* enforcer. Because effective enforcers are transparent, they may transform valid input sequences only into semantically equivalent output sequences, for some system-specific definition of semantic equivalence. When two executions $\sigma, \tau \in \mathcal{A}^\infty$ are semantically equivalent, we write $\sigma \approx \tau$. We place no restrictions on a relation of semantic equivalence except that it actually be an equivalence relation (i.e., reflexive, symmetric, and transitive), and that properties of interest $\hat{P}$ do not distinguish between semantically equivalent executions.

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

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

Definition 2.3 *Effective$_\approx$ Enforcement.* An automaton S with starting state q_0 *effectively$_\approx$ enforces* a property $\hat{P}$ on a system with action set $\mathcal{A}$ and semantic equivalence relation $\approx$ if and only if $\forall \sigma \in \mathcal{A}^\infty : \exists \tau \in \mathcal{A}^\infty$:

(1) $(q_0, \sigma) \Downarrow_S \tau$,

(2) $\hat{P}(\tau)$, and

(3) $\hat{P}(\sigma) \implies \sigma \approx \tau$

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

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

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

Definition 2.4 *Effective$_=$ Enforcement.* An automaton S with starting state q_0 *effectively$_=$ enforces* a property $\hat{P}$ on a system with action set $\mathcal{A}$ if and only if $\forall \sigma \in \mathcal{A}^\infty : \exists \tau \in \mathcal{A}^\infty$:

(1) $(q_0, \sigma) \Downarrow_S \tau$,

(2) $\hat{P}(\tau)$, and

(3) $\hat{P}(\sigma) \implies \sigma{=}\tau$

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

3. POLICIES ENFORCEABLE BY MONITORS

Now that we have set up a framework for formally reasoning about policies, properties, monitors (security automata), and enforcement, we can consider the space of properties enforceable by program monitors. In this section, we examine the enforcement powers of two types of monitors: a very simple but widely studied variety that we model with *truncation automata* (Section 3.1) and a more sophisticated variety that we model with *edit automata* (Section 3.2). In Section 4, we compare the properties enforceable by these two types of monitors and show that although the simple monitors can enforce only safety properties, it is possible to enforce some nonsafety properties using more sophisticated monitors.

3.1 Truncation Automata

We begin by demonstrating why it is often believed that program monitors enforce only safety properties: this belief is provably correct when considering a common but very limited type of monitor that we model by *truncation automata*. A truncation automaton has only two options when it intercepts an action from the target program: it may accept the action and make it observable, or it may halt (i.e., truncate the action sequence of) the target program altogether. Schneider first defined this model of program monitors [Schneider 2000], and other authors have similarly focused on this simple, though limited, model when considering the properties enforceable by security automata [Viswanathan 2000; Kim et al. 2002; Fong 2004]. Truncation-based monitors have been used successfully to enforce a rich set of interesting safety policies including access control [Evans and Twyman 1999], stack inspection [Erlingsson and Schneider 1999; Abadi and Fournet 2003], software fault isolation [Wahbe et al. 1993; Erlingsson and Schneider 2000], Chinese Wall [Brewer and Nash 1989; Erlingsson 2003; Fong 2004], and one-out-of-k authorization [Fong 2004] policies.[3]

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

3.1.1 *Definition.* A truncation automaton T is a finite or countably infinite state machine (Q, q_0, δ) that is defined with respect to some system with action set $\mathcal{A}$. As usual, Q specifies the possible automaton states, and q_0 is the initial state. The complete function $\delta : Q \times \mathcal{A} \rightarrow Q \cup \{halt\}$ specifies the transition function for the automaton and indicates either that the automaton should accept the current

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

input action and move to a new state (when the return value is a new state), or that the automaton should halt the target program (when the return value is *halt*). For the sake of determinacy, we require that *halt* $\notin Q$. The operational semantics of truncation automata are formally specified by the following rules.

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

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

if $\sigma = a;\sigma'$
and $\delta(q,a) = q'$

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

if $\sigma = a;\sigma'$
and $\delta(q,a) = \mathit{halt}$

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

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

When given as input some $\sigma \in \mathcal{A}^\infty$ such that $\hat{P}(\sigma)$, a truncation automaton that effectively$_{=}$ enforces $\hat{P}$ must output σ. However, the automaton must also truncate every invalid input sequence into a valid output. Any truncation of an invalid input prevents the automaton from accepting all the actions in a valid extension of that input. Therefore, truncation automata cannot effectively$_{=}$ enforce any property in which an invalid execution can be a prefix of a valid execution. This is exactly the definition of safety properties, so it is clear that truncation automata effectively$_{=}$ enforce only safety properties.

Past research claimed to equate the enforcement power of truncation automata with the set of computable safety properties [Viswanathan 2000; Kim et al. 2002]. We improve previous work by showing that there is exactly one computable safety property unenforceable by any sound security automaton: the unsatisfiable safety property that considers all executions invalid. A monitor in our framework cannot enforce such a property because there is no valid output sequence that could be produced in response to an invalid input sequence. To prevent this case and to ensure that truncation automata can behave correctly on targets that generate no actions, we require that the empty sequence satisfies any property we are interested in enforcing. We often use the term *reasonable* to describe computable properties $\hat{P}$ such that $\hat{P}(\cdot)$.

Definition 3.1 *Reasonable Property.* A property $\hat{P}$ on a system with action set $\mathcal{A}$ is *reasonable* if and only the following conditions hold.

(1) $\hat{P}(\cdot)$

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

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

THEOREM 3.2 EFFECTIVE$_=$ T^∞-ENFORCEMENT. *A property $\hat{P}$ on a system with action set $\mathcal{A}$ can be effectively$_=$ enforced by some truncation automaton T if and only if the following constraints are met.*

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

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

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

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

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

Let $\sigma \in \mathcal{A}^\infty$ be the input to T. If $\neg\hat{P}(\sigma)$ then by the safety condition in the theorem statement, $\exists\sigma'\preceq\sigma.\neg\hat{P}(\sigma')$. By $I_{\hat{P}}(\sigma')$, T can never enter the state for this σ' and must therefore halt on input σ. Let τ be the final state reached on input σ. By $I_{\hat{P}}(\tau)$ and the fact that T halts (ceases to make transitions) after reaching state τ, we have $\hat{P}(\tau)$ and $(q_0, \sigma) \Downarrow_T \tau$.

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

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

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

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

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

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

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

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

Theorem 3.3 Effective$_\approx$ T^∞-Enforcement. *A property $\hat{P}$ on a system with action set $\mathcal{A}$ can be effectively$_\approx$ enforced by some truncation automaton T if and only if there exists a decidable predicate D over $\mathcal{A}^\star$ such that the following constraints are met.*

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

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

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

This transition function is computable because D is decidable.

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

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

We split the case where $\hat{P}(\sigma)$ into two subcases. If T never truncates input σ then T outputs every prefix of σ and only prefixes of σ, so by the definition of $\Downarrow_T$, $(q_0, \sigma) \Downarrow_T \sigma$. Because $\hat{P}(\sigma)$ and $\sigma \approx \sigma$, T effectively$_\approx$ enforces $\hat{P}$ in this subcase. On the other hand, if T truncates input σ, it does so in some state σ' while making a transition on action a (hence, $\sigma'; a\preceq\sigma$) because $D(\sigma'; a)$. In this subcase, $I_{\hat{P}}(\sigma')$ implies $(q_0, \sigma) \Downarrow_T \sigma'$. Also, since $D(\sigma'; a)$ forced T to halt, constraint 2 in the theorem statement ensures that $\hat{P}(\sigma')$ and $\sigma' \approx \sigma$. Therefore, T correctly effectively$_\approx$ enforces $\hat{P}$ in all cases.

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

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

On practical systems, it is likely uncommon that the property requiring enforcement and the system's relation of semantic equivalence are so broadly defined that some invalid execution has a prefix that not only can be extended to a valid execution, but that is also equivalent to *all* valid extensions of the prefix. We therefore consider the set of properties detailed in the theorem of Effective$_=$ T^∞-Enforcement (i.e., reasonable safety properties) more indicative of the true enforcement power of truncation automata.

3.2 Edit Automata

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

3.2.1 *Definition.* An *edit automaton* E is a triple (Q, q_0, δ) defined with respect to some system with action set $\mathcal{A}$. As with truncation automata, Q is the possibly countably infinite set of states, and q_0 is the initial state. In contrast to truncation automata, the complete transition function δ of an edit automaton has the form $\delta : Q \times \mathcal{A} \to Q \times (\mathcal{A} \cup \{\cdot\})$. The transition function specifies, when given a current state and input action, a new state to enter and either an action to *insert* into the output stream (without consuming the input action) or the empty sequence to indicate that the input action should be *suppressed* (i.e., consumed from the input without being made observable). In other work, we have defined edit automata that can additionally perform the following transformations in a single step: insert a finite sequence of actions, accept the current input action, or halt the target [Ligatti et al. 2005a]. However, all of these transformations can be expressed in terms of suppressing and inserting single actions. For example, an edit automaton can halt a target by suppressing all future actions of the target; an edit automaton accepts an action by inserting and then suppressing that action (first making the action observable and then consuming it from the input). Although in practice these transformations would each be performed in a single step, we have found the minimal operational semantics containing only the two rules shown below more amenable to formal reasoning. Explicitly including the additional rules in the model would not invalidate any of our results.

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

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

if $\sigma = a; \sigma'$
and $\delta(q, a) = (q', a')$

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

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

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

3.2.2 *Enforceable Properties.* Edit automata are powerful property enforcers because they can suppress a sequence of potentially illegal actions and later, if the sequence is determined to be legal, just re-insert it. Essentially, the monitor feigns to the target that its requests are being accepted, although none actually are, until the monitor can confirm that the sequence of feigned actions is valid. At that point, the monitor inserts all of the actions it previously feigned accepting. This is the same idea implemented by intentions files in database transactions [Paxton 1979]. Monitoring systems like virtual machines can also be used in this way, feigning execution of a sequence of the target's actions and only making the sequence observable when it is known to be valid.

As we did for truncation automata, we develop a lower bound on the set of properties that edit automata effectively$_{\approx}$ enforce by considering the properties they effectively$_=$ enforce. Using the above-described technique of suppressing invalid inputs until the monitor determines that the suppressed input obeys a property, edit automata can effectively$_=$ enforce any reasonable *infinite renewal* (or simply *renewal*) property. A renewal property is one in which every valid infinite-length sequence has infinitely many valid prefixes, and conversely, every invalid infinite-length sequence has only finitely many valid prefixes. For example, a property $\hat{P}$ may be satisfied only by executions that contain the action a. This is a renewal property because valid infinite-length executions contain an infinite number of valid prefixes (in which a appears) while invalid infinite-length executions contain only a finite number of valid prefixes (in fact, zero). This $\hat{P}$ is also a liveness property because any invalid finite execution can be made valid simply by appending the action a. Although edit automata cannot enforce this $\hat{P}$ because $\neg\hat{P}(\cdot)$, in Section 4.2 we will recast this example as a reasonable "eventually audits" policy and show several more detailed examples of renewal properties enforceable by edit automata.

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

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

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

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

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

Theorem 3.4 Lower Bound Effective$_=$ E^∞-Enforcement. *A property $\hat{P}$ on a system with action set $\mathcal{A}$ can be effectively$_=$ enforced by some edit automaton E if the following constraints are met.*

(1) $\forall\sigma \in \mathcal{A}^\omega : \hat{P}(\sigma) \iff (\forall\sigma' \preceq \sigma : \exists\tau \preceq \sigma : \sigma' \preceq \tau \wedge \hat{P}(\tau))$ $\qquad (\text{Renewal}_2)$
(2) $\hat{P}(\cdot)$
(3) $\forall\sigma \in \mathcal{A}^\star : \hat{P}(\sigma)$ *is decidable*

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

—States: $Q = \mathcal{A}^\star \times \mathcal{A}^\star \times \{0, 1\}$ (the sequence of actions output so far, the sequence of actions currently suppressed, and a flag indicating whether the suppressed actions need to be inserted)

—Start state: $q_0 = (\cdot, \cdot, 0)$ (nothing has been output or suppressed)

—Transition function:

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

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

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

Let $\sigma \in \mathcal{A}^\infty$ be the input to the automaton E. If $\neg\hat{P}(\sigma)$ and $\sigma \in \mathcal{A}^\star$ then by the automaton invariant, E consumes all of input σ and halts in some state $(\tau,\sigma',0)$ such that $(q_0,\sigma) \Downarrow_E \tau$ and $\hat{P}(\tau)$. Hence, E effectively$_=$ enforces $\hat{P}$ in this case. If $\neg\hat{P}(\sigma)$ and $\sigma \in \mathcal{A}^\omega$ then by the renewal condition in the theorem statement, there must be some prefix σ' of σ such that for all longer prefixes τ of σ, $\neg\hat{P}(\tau)$. Thus, by the transition function of E, the invariant of E, and the definition of $\Downarrow_E$, E on input σ outputs only some finite τ' such that $\hat{P}(\tau')$ and $(q_0,\sigma) \Downarrow_E \tau'$ (and E suppresses all actions in σ after outputting τ').

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

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

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

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

THEOREM 3.5 EFFECTIVE$_=$ E^∞-ENFORCEMENT. *A property $\hat{P}$ on a system with action set $\mathcal{A}$ can be effectively$_=$ enforced by some edit automaton E if and only if the following constraints are met.*

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

(2) $\hat{P}(\cdot)$

(3) $\forall\sigma \in \mathcal{A}^\star : \hat{P}(\sigma)$ *is decidable*

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

—States: $Q = \mathcal{A}^\star \times \mathcal{A}^\star$ (the sequence of actions output so far paired with the sequence of actions suppressed since the previous insertion)

—Start state: $q_0 = (\cdot, \cdot)$ (nothing has been output or suppressed)

—Transition function (for simplicity, we abbreviate δ):
Consider processing the action a in state (τ', σ').
(A). If we can compute from $\tau';\sigma'$ the existence and actions of some $\sigma \in \mathcal{A}^\omega$ such that $\forall\tau{\succeq}(\tau';\sigma') : \hat{P}(\tau) \implies \tau{=}\sigma$, enter an infinite loop that inserts one by one all actions necessary to output every prefix of σ.
(B). Otherwise, if $\hat{P}(\tau';\sigma';a)$ then insert $\sigma';a$ (one action at a time), suppress a, and continue in state $(\tau';\sigma';a, \cdot)$.
(C). Otherwise, suppress a and continue in state $(\tau', \sigma';a)$.

This automaton is an informal version of the one constructed in the "if" direction of the proof of Theorem 3.4, except for the addition of transition (A), and E effectively$_=$ enforces $\hat{P}$ for the same reasons given there. The only difference is that E can insert an infinite sequence of actions if it computes that only that sequence of actions can extend the current input to satisfy $\hat{P}$. In this case, E continues to effectively$_=$ enforce $\hat{P}$ because its output satisfies $\hat{P}$ and equals any valid input sequence.

(Only-If Direction). Consider any $\sigma \in \mathcal{A}^\omega$ such that $\hat{P}(\sigma)$. By the definition of effective$_=$ enforcement, $(q_0, \sigma) \Downarrow_E \sigma$, where q_0 is the initial state of E. By the definitions of $\Downarrow_E$ and $=$, E must output all prefixes of σ and only prefixes of σ when σ is input. Assume for the sake of obtaining a contradiction that the extended

renewal constraint is untrue for σ. This implies that there is some valid prefix σ' of σ after which all longer prefixes of σ violate $\hat{P}$. After outputting σ' on input σ', E cannot output any prefix of σ without outputting every prefix of σ (if it did, its output would violate $\hat{P}$). But because the extended renewal constraint does not hold on σ by assumption, either (1) more than one valid execution will always extend the automaton's input or (2) E can never compute or emit all prefixes of σ. Therefore, E cannot output every prefix of σ after outputting σ', so E fails to effectively$_=$ enforce $\hat{P}$ on this σ. Our assumption was therefore incorrect, and the renewal constraint must hold.

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

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

Finally, we decide $\hat{P}(\sigma)$ for all $\sigma \in \mathcal{A}^\star$ using the same procedure described in the "Only-If" direction of the proof of Theorem 3.2. $\square$

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

THEOREM 3.6 EFFECTIVE$_\approx$ E^∞-ENFORCEMENT. *Let* repls *be a decidable function* repls : $\mathcal{A}^\star \times \mathcal{A}^\star \rightarrow \mathcal{A} \cup \{\cdot\}$. *Then* $R^\star_{\text{repls}}$ *is a decidable function* $R^\star_{\text{repls}} : \mathcal{A}^\star \times \mathcal{A}^\star \times \mathcal{A}^\star \rightarrow \mathcal{A}^\star$ *parameterized by* repls *and inductively defined as follows, where all metavariables are universally quantified.*

—$R^\star_{\text{repls}}(\cdot, \sigma, \tau) = \tau$
—$(\text{repls}(\sigma; a, \tau) = \cdot) \implies R^\star_{\text{repls}}(a; \sigma', \sigma, \tau') = R^\star_{\text{repls}}(\sigma', \sigma; a, \tau')$
—$(\text{repls}(\sigma; a, \tau) = a') \implies R^\star_{\text{repls}}(a; \sigma', \sigma, \tau') = R^\star_{\text{repls}}(a; \sigma', \sigma, \tau'; a')$

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

(1) $\forall \sigma' \preceq \sigma : \forall \tau' \in \mathcal{A}^\star : (R^\star_{\text{repls}}(\sigma', \cdot, \cdot) = \tau') \implies \tau' \preceq \tau$

(2) $\forall \tau' \preceq \tau : \exists \sigma' \preceq \sigma : R^{\star}_{\text{repls}}(\sigma', \cdot, \cdot) = \tau'$

(3) $\hat{P}(\tau)$

(4) $\hat{P}(\sigma) \implies \sigma \approx \tau$

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

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

—States: $Q = \mathcal{A}^{\star} \times \mathcal{A}^{\star}$ (the input processed and the output emitted so far)

—Start state: $q_0 = (\cdot, \cdot)$ (nothing processed or output)

—Transition function: $\delta((\sigma, \tau), a) = \begin{cases} ((\sigma, \tau; a'), a') & \text{if } \text{repls}(\sigma; a, \tau) = a' \\ ((\sigma; a, \tau), \cdot) & \text{otherwise} \end{cases}$

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

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

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

As with truncation automata, we believe that the theorems related to edit automata acting as effective$_{=}$ enforcers more naturally capture their inherent power than does the theorem of effective$_{\approx}$ enforcement. Effective$_{=}$ enforcement provides an elegant lower bound for what can be effectively$_{\approx}$ enforced in practice.

Limitations. In addition to standard assumptions of program monitors, such as that a target cannot circumvent or corrupt a monitor, our theoretical model makes assumptions particularly relevant to edit automata that are sometimes violated in practice. Most importantly, our model assumes that security automata have the same computational capabilities as the system that observes the monitor's output. If an action violates this assumption by requiring an outside system in order to be executed, it cannot be feigned (i.e., suppressed) by the monitor. For example, it would be impossible for a monitor to feign sending email, wait for the target to receive a response to the email, test whether the target does something invalid with the response, and then decide to undo sending email in the first place. Here, the action for sending email has to be made observable to systems outside of the monitor's control in order to be executed, so this is an unsuppressible action. A similar limitation arises with time-dependent actions, where an action cannot be feigned (i.e., suppressed) because it may behave differently if made observable later. In addition to these sorts of unsuppressible actions, a system may contain actions uninsertable by monitors because, for example, the monitors lack access to secret keys that must be passed as parameters to the actions. In the future, we plan to explore the usefulness of including sets of unsuppressible and uninsertable actions in the specification of systems. We might be able to harness some of our other work [Ligatti et al. 2005a], which defined security automata limited to inserting (insertion automata) or suppressing (suppression automata) actions, toward this goal.

4. INFINITE RENEWAL PROPERTIES

In this section, we examine some interesting aspects of the class of infinite renewal properties. We compare renewal properties to safety and liveness properties and provide several high-level examples of renewal properties that are neither safety nor liveness properties. Section 5 contains lower-level examples of concrete nonsafety policies that we have enforced in an implemented system.

4.1 Renewal, Safety, and Liveness

The most obvious way in which safety and infinite renewal properties differ is that safety properties place restrictions on finite executions (invalid finite executions must have some prefix after which all extensions are invalid), while renewal properties place no restrictions on finite executions. Thus, if we consider systems that only exhibit finite executions, edit automata can enforce *every* reasonable property [Ligatti et al. 2005a]. Without infinite-length executions, every property is a renewal property.

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

There are renewal properties that are not liveness properties (e.g., the property that is only satisfied by the empty sequence), and there are liveness properties that are not renewal properties (e.g., the nontermination property only satisfied by infinite executions). Some renewal properties, such as the one only satisfied by the empty sequence and the sequence $a;a$, are neither safety nor liveness. Although Alpern and Schneider [Alpern and Schneider 1985] showed that exactly one property is both safety and liveness (the property satisfied by every execution), some interesting liveness properties are also renewal properties. We examine examples of such renewal properties in the following subsection.

4.2 Example Properties

We next present several examples of renewal properties that are not safety properties, as well as some examples of nonrenewal properties. By the theorems in Sections 3.1.2 and 3.2.2, the nonsafety renewal properties are effectively$_=$ enforceable by edit automata but not by truncation automata. Moreover, the proof of Theorem 3.4 shows how to construct an edit automaton to enforce any of the renewal properties described in this subsection. Later, in Section 5, we examine additional nonsafety properties and show how they can be specified and enforced using practical program monitors.

Renewal properties. Suppose we wish to constrain a user's interaction with a computer system. A user may execute any sequence of actions that does not involve opening files but must eventually log out. The process of executing non-file-open actions and then logging out may repeat indefinitely, so we might write the requisite property $\hat{P}$ more specifically as $(a_1{}^\star; a_2)^\infty$, where a_2 ranges over all actions for logging out, a_3 over actions for opening files, and a_1 over all other actions.[4] This $\hat{P}$ is not a safety property because a finite sequence of only a_1 events disobeys $\hat{P}$ but can be extended (by appending a_2) to obey $\hat{P}$. Our $\hat{P}$ is also not a liveness property because there are finite executions that cannot be extended to satisfy $\hat{P}$, such as the sequence containing only a_3. However, this nonsafety, nonliveness property is a renewal property because infinite-length executions are valid if and only if they contain infinitely many (valid) prefixes of the form $(a_1{}^\star; a_2)^\star$.

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

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

As briefly mentioned in Section 3.2.2, edit automata derive their power from being able to operate in a way similar to intentions files in database transactions. At a high level, any transaction-based property is a renewal property. Let τ range over

[4]As Alpern and Schneider note [Alpern and Schneider 1985], this sort of $\hat{P}$ might be expressed with the (strong) *until* operator in temporal logic; event a_1 occurs *until* event a_2.

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

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

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

Figure 1 summarizes the results of the preceding discussion and that of Section 4.1. The Trivial property in Figure 1 considers all executions legal and is the only property in the intersection of safety and liveness properties.

5. ENFORCING NONSAFETY POLICIES WITH PRACTICAL MONITORS

Section 4 showed that program monitors modeled by edit automata can enforce some nonsafety properties and provided high-level examples of such properties. This section details lower-level nonsafety monitoring policies that we have implemented and enforced; these examples demonstrate the practical applicability of our

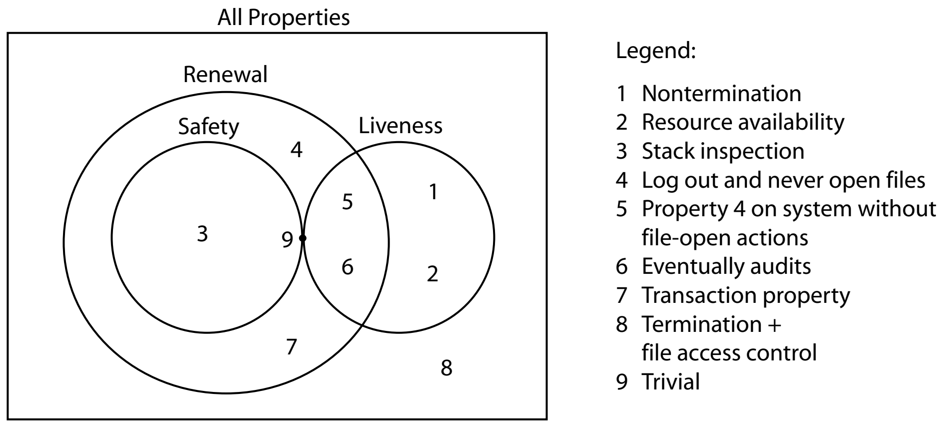

Fig. 1. Relationships between safety, liveness, and renewal properties.

theoretical analysis.

In Section 5.1, we provide an overview of Polymer, the system in which we have implemented concrete nonsafety policies. This overview should suffice for understanding the basic concepts underlying the example policies presented in Section 5.2. We refer interested readers to other papers for more details regarding the Polymer language and enforcement system [Bauer et al. 2005a; Ligatti 2006].

5.1 Polymer Overview

Polymer is a language and system for enforcing run-time policies. It allows users to specify policies describing the behavior of Java programs, and it compiles those policies into Java bytecode monitors. When users execute untrusted Java applications, Polymer rewrites the class files used by the untrusted applications to invoke the previously compiled monitors. This technique of bytecode (or binary) rewriting to invoke run-time monitors is a common implementation strategy [Erlingsson 2003; Evans 2000; Hamlen 2006].

A user specifies a Polymer policy by writing a new class that extends Polymer's default `Policy` class. In the new policy class, the user defines two key methods: `public Sug query(Action a)` and `public void accept(Sug s)`.

The `query` method gets invoked immediately before a monitored method (called the *trigger action*) executes, allowing the monitor to intervene and suppress or insert actions. The `Action` object passed to the `query` method describes the about-to-be-executed trigger action; the `Sug` (i.e., suggestion) object returned by the `query` method is the monitor's response to that trigger action. The types of suggestions a monitor can make include `InsSug` (which inserts an action), `ReplSug` (which suppresses an action by replacing it with some precomputed return value), `HaltSug` (which suppresses all future actions of the untrusted application by halting it entirely), and `IrrSug` and `OKSug` (which both allow the trigger action to proceed undisturbed).

The `accept` method in a Polymer policy performs state updates after the `query` method has executed and returned a suggestion object. The separation of `accept` from `query` methods relates to Polymer's ability to compose policies (for more details, see [Bauer et al. 2005a]).

5.2 Example Nonsafety Policies Enforceable in Polymer

Because Polymer implements all the ways in which edit automata can react to trigger actions—`InsSug` *inserts* an action, and `ReplSug` and `HaltSug` are different ways to *suppress* actions (and `IrrSug` and `OKSug` are different ways to *accept* actions)—we can use the Polymer system to enforce some nonsafety properties. For the remainder of this section we will study two reasonable and simple examples of nonsafety policies enforceable in Polymer. The first example ensures that (hypothetical) ATM machines generate a proper log when dispensing cash, while the second example ensures that targets writing to a file eventually give the file satisfactory contents (e.g., that the file obeys the required file format).

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

(1) `logBegin(`n`)` creates a log message that the ATM is about to dispense n dollars.
(2) `dispense(`n`)` causes the ATM to dispense n dollars.
(3) `logEnd(`n`)` creates a log message that the ATM just completed dispensing n dollars.

Suppose we wish to require that the ATM machine's (frequently updated and occasionally buggy) software properly logs all cash dispensations; we will consider an execution valid if and only if it has the form $(\texttt{logBegin}(n); \texttt{dispense}(n); \texttt{logEnd}(n))^\infty$. That is, valid executions must be sequences of valid transactions, where each valid transaction consists of logging that some amount of cash is about to be dispensed, dispensing that cash, and then logging that that amount of cash has just been dispensed. Our desired policy is a nonsafety, nonliveness, renewal property. It is nonsafety because there exists an invalid execution ($\texttt{logBegin}(20)$) that prefixes a valid execution ($\texttt{logBegin}(20); \texttt{dispense}(20); \texttt{logEnd}(20)$). It is nonliveness because some invalid execution ($\texttt{dispense}(20)$) cannot be made valid through extension. Nonetheless, this nonsafety, nonliveness property is clearly a transaction-style renewal property (as described in Section 4.2).

We can enforce this nonsafety policy in Polymer by suppressing preliminary `logBegin` and `dispense` actions until we are guaranteed that the current transaction is valid, at which point the suppressed actions get re-inserted. Figure 2 contains the `AtmPolicy`, which employs this enforcement technique and suppresses actions simply by returning `ReplSug`s in the `query` method. For simplicity, we assume in the example code that all re-inserted methods complete normally. This frees us from worrying about exceptions being raised and program-terminating actions being called during the inserted methods, which could prevent our policy from inserting all of the actions necessary to satisfy the property. We can remove this simplifying assumption by modifying our policy so that it catches exceptions raised (and `done` actions invoked) during execution of inserted actions but does not allow the exceptions to propagate or the virtual machine to exit until all required insertions have been made.

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

    public Sug query(Action a) {
        aswitch (a) {
        case ⟨void examples.ATM.logBegin(int n)⟩:
            if (transState==0) return new ReplSug(null, a);
            else return new HaltSug(a);
        case ⟨void examples.ATM.dispense(int n)⟩:
            if (transState==1 && amt==n) return new ReplSug(null, a);
            else return new HaltSug(a);
        case ⟨void examples.ATM.logEnd(int n)⟩:
            if (transState==2 && amt==n) return new OKSug(a);
            else return new HaltSug(a);
        default :
            if (transState ⟩ 0) return new HaltSug(a);
            else return new IrrSug();
        }
    }

    public void accept(Sug s) {
        aswitch (s.getTrigger()) {
            case ⟨void examples.ATM.logBegin(int n)⟩: transState = 1; amt = n; break ;
            case ⟨void examples.ATM.dispense(int n)⟩: transState = 2;
        }
        if (s.isOK()) {                    // Transaction is valid.
            isInsert = true;
            examples.ATM.logBegin(amt); examples.ATM.dispense(amt);
            isInsert = false;
            amt = 0; transState = 0;
        }
    }
}
```

Fig. 2. Nonsafety Polymer policy ensuring that ATM cash dispensation gets logged properly.

File-contents Policy. Let us consider enforcing a property that allows files to be written, possibly using multiple file-write operations, if and only if the file contents eventually satisfy some predicate that is passed to the policy as a parameter. This predicate could be satisfied, for example, if the file ends with an appropriate copyright notice or disclaimer. This predicate might not hold in the middle of a sequence of writes but could be satisfied after a later write, and must hold at the end of a sequence of file writes to ensure that every file on the system carries the proper copyright notice. This requirement that a file contain a copyright notice is not a safety property: we can overwrite a file's copyright notice with other data, making the execution invalid; however, we can extend the invalid execution to satisfy the property by appending the proper copyright text to the file. Actually, if we assume that the file predicate is satisfiable then this file-contents policy is a liveness prop-

```
public class FilePredPolicy extends Policy {
    // FilePrediate has a method for testing whether a file's contents are OK.
    private FilePredicate filePred;
    // Store whether file writes are due to our inserting them.
    private boolean areWeInserting = false;

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

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

    public void accept(Sug s) {
        aswitch (s.getTrigger()) {
            case ⟨abs void absact.FileWrite(byte [ ] b, int off, int len)⟩:
                // We are suppressing a write; next insert all suppressed writes that need to be
                // inserted (performInsertions uses filePred to determine what must be inserted).
                areWeInserting = true;
                performInsertions(s.getTrigger());
                areWeInserting = false;
        }
    }
}
```

Fig. 3. Abbreviated nonsafety Polymer policy ensuring that files are written satisfactorily.

erty: any invalid finite execution becomes valid when the target executes whatever file-write operations satisfy the file predicate.

As expected, we enforce this nonsafety, liveness, file-contents property by suppressing (feigning) writes to files until we can ensure their validity, at which point we re-insert all suppressed writes to the now-valid file. Figure 3 contains an abbreviated Polymer policy that enforces this file-contents property. In its constructor, `FilePredPolicy` accepts an object that implements the `FilePredicate` interface, which contains a method to test file validity when given a file name and a sequence of writes to that file. The `FilePredPolicy` uses the `FilePredicate` in its `performInsertions` method to check whether to insert a file's suppressed writes. The policy remembers which writes have been suppressed for each file by maintaining a mapping from files to ordered lists of pending write operations. Maintenance of this mapping occurs in the `performInsertions` method. In order to properly feign file write operations, `FilePredPolicy` also monitors actions that read data associated with files (i.e., file contents, lengths, and modification times), and ensures that the target sees files as if suppressed writes have actually executed. Hence, our Polymer monitor effectively enforces the file-contents policy: the monitor does not modify valid executions' semantics (though intermediate file writes may be performed later than they normally would), and the monitor ensures that all observed executions are valid.

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

6. CONCLUSIONS

This article improves our understanding of the space of policies program monitors can enforce. We conclude by summarizing our primary contributions (Section 6.1), enumerating some directions for future work (Section 6.2), and making closing remarks (Section 6.3).

6.1 Summary

As outlined in Section 1.2, we have made four principal contributions. First, we have created a framework for reasoning about run-time policy enforcement. The framework makes explicit all of our assumptions about what constitutes a policy, a monitor, and enforcement of a policy by a monitor. Second, we have applied the framework to delineate the policies enforceable by two models of monitors, finding that although simple monitors enforce exactly the set of reasonable safety properties, sophisticated monitors enforce exactly the set of infinite renewal properties, which we have introduced. Third, we have analyzed the set of renewal properties and found that it contains some nonsafety (and even some liveness) properties; hence, monitors can enforce some nonsafety properties. Fourth, we have validated our formal analysis by demonstrating concrete nonsafety policies we have implemented and enforced with run-time monitors in the Polymer system.

6.2 Future Work

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

Practical Constraints on Theoretical Monitors. Sections 3.2.2 and 5 discuss a practical limitation of monitors absent from our current theoretical model: monitors often do not have the same computational capabilities as the machine that executes target actions. This limitation of real monitors implies that some actions cannot be suppressed (i.e., the monitor cannot "feign" an action), and some actions cannot be inserted (i.e., the monitor cannot obtain information needed to invoke an action). One easily imagined extension of our current framework is to incorporate sets of unsuppressible and uninsertable actions into system definitions and to analyze which properties edit automata can enforce under those conditions. This extension would make our model more precise, though significantly more complex.

Several additional practical constraints could be placed on monitors For instance, Fong has shown that limiting the memory available to monitors induces limits on the properties they can enforce [Fong 2004]. We might ask similar questions of time bounds on monitors: Are there useful properties that require super-polynomial monitoring time to enforce? How can we add real-time constraints to our model to reflect practical limits on the amount of real time monitors may consume, and how do these constraints affect the enforcement of real-time policies?

Formally Linking Edit Automata with Polymer Policies. Section 5.2 included an informal description of the ability of Polymer policies to implement edit automata: Polymer's `InsSug` *inserts* an action, and its `ReplSug` and `HaltSug` *suppress* actions (and its `IrrSug` and `OKSug` *accept* actions). This implementation is simple and intuitive, but it would be nice to have a formally proven bisimulation between the operational semantics of edit automata and the policies expressible in a system like Polymer. Proving such a bisimulation would be interesting because it would tie practical monitor specifications to properties enforceable by edit automata, allowing us to describe formally the space of policies enforceable in practice.

6.3 Closing Remarks

Given their abundance and practicality as enforcement mechanisms, it seems strange that we understand relatively little of monitors' actual enforcement capabilities and have relatively primitive tools for designing, reasoning about, and implementing monitors. Even basic results, such as that practical monitors can sometimes enforce liveness properties (Section 5), surprise us.

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

REFERENCES

Abadi, M. and Fournet, C. 2003. Access control based on execution history. In *Proceedings of the 10th Annual Network and Distributed System Symposium.*

Alpern, B. and Schneider, F. 1987. Recognizing safety and liveness. *Distributed Computing 2*, 117–126.

Alpern, B. and Schneider, F. B. 1985. Defining liveness. *Information Processing Letters 21,* 4 (Oct.), 181–185.

Bauer, L., Ligatti, J., and Walker, D. 2002. More enforceable security policies. In *Foundations of Computer Security.* Copenhagen, Denmark.

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

Bauer, L., Ligatti, J., and Walker, D. 2005a. Composing security policies with polymer. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*. Chicago.

Bauer, L., Ligatti, J., and Walker, D. 2005b. Polymer: A language for composing run-time security policies. `http://www.cs.princeton.edu/sip/projects/polymer/`.

Biba, K. J. 1975. Integrity considerations for secure computer systems. Tech. Rep. ESD-TR-76-372, MITRE Corporation. July.

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

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

Damianou, N., Dulay, N., Lupu, E., and Sloman, M. 2001. The Ponder policy specification language. *Lecture Notes in Computer Science 1995*, 18–39.

Edjlali, G., Acharya, A., and Chaudhary, V. 1998. History-based access control for mobile code. In *ACM Conference on Computer and Communications Security*. 38–48.

Erlingsson, Ú. 2003. The inlined reference monitor approach to security policy enforcement. Ph.D. thesis, Cornell University.

Erlingsson, Ú. and Schneider, F. B. 1999. SASI enforcement of security policies: A retrospective. In *Proceedings of the New Security Paradigms Workshop*. Caledon Hills, Canada, 87–95.

Erlingsson, Ú. and Schneider, F. B. 2000. IRM enforcement of Java stack inspection. In *IEEE Symposium on Security and Privacy*. Oakland, CA.

Evans, D. 2000. Policy-directed code safety. Ph.D. thesis, Massachusetts Institute of Technology.

Evans, D. and Twyman, A. 1999. Flexible policy-directed code safety. In *IEEE Security and Privacy*. Oakland, CA.

Fong, P. W. L. 2004. Access control by tracking shallow execution history. In *IEEE Symposium on Security and Privacy*. Oakland, CA.

Hamlen, K. 2006. Security policy enforcement by automated program-rewriting. Ph.D. thesis, Cornell University.

Hamlen, K., Morrisett, G., and Schneider, F. B. 2006. Computability classes for enforcement mechanisms. *ACM Transactions on Progamming Languages and Systems 28,* 1 (Jan.), 175–205.

Havelund, K. and Roşu, G. 2004. Efficient monitoring of safety properties. *International Journal on Software Tools for Technology Transfer (STTT) 6,* 2 (Aug.), 158–173.

Jeffery, C., Zhou, W., Templer, K., and Brazell, M. 1998. A lightweight architecture for program execution monitoring. In *Program Analysis for Software Tools and Engineering (PASTE)*. ACM Press, 67–74.

Kim, M., Kannan, S., Lee, I., Sokolsky, O., and Viswantathan, M. 2002. Computational analysis of run-time monitoring—fundamentals of Java-MaC. In *Run-time Verification*.

Kim, M., Viswanathan, M., Ben-Abdallah, H., Kannan, S., Lee, I., and Sokolsky, O. 1999. Formally specified monitoring of temporal properties. In *European Conference on Real-time Systems*. York, UK.

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

Liao, Y. and Cohen, D. 1992. A specificational approach to high level program monitoring and measuring. *IEEE Trans. Softw. Eng. 18,* 11, 969–978.

Ligatti, J. 2006. Policy enforcement via program monitoring. Ph.D. thesis, Princeton University.

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

Ligatti, J., Bauer, L., and Walker, D. 2005a. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security 4,* 1–2 (Feb.), 2–16.

LIGATTI, J., BAUER, L., AND WALKER, D. 2005b. Enforcing non-safety security policies with program monitors. In *10th European Symposium on Research in Computer Security (ESORICS)*. Milan, Italy.

LYNCH, N. A. AND TUTTLE, M. R. 1987. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the 6th annual ACM Symposium on Principles of Distributed Computing*. ACM Press, 137–151.

PAXTON, W. H. 1979. A client-based transaction system to maintain data integrity. In *Proceedings of the 7th ACM symposium on Operating Systems Principles*. ACM Press, 18–23.

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

RUSSINOVICH, M. 2005. Sony, rootkits and digital rights management gone too far. http://www.sysinternals.com/blog/2005/10/sony-rootkits-and-digital-rights.html.

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

SEN, K., VARDHAN, A., AGHA, G., AND ROSU, G. 2004. Efficient decentralized monitoring of safety in distributed systems. In *26th International Conference on Software Engineering (ICSE'04)*. 418–427.

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

WAHBE, R., LUCCO, S., ANDERSON, T., AND GRAHAM, S. 1993. Efficient software-based fault isolation. In *Fourteenth ACM Symposium on Operating Systems Principles*. Asheville, 203–216.