Enforcing Non-safety Security Policies with Program Monitors

Jay Ligatti[1], Lujo Bauer[2], and David Walker[1]

[1] Department of Computer Science, Princeton University
[2] CyLab, Carnegie Mellon University

Abstract. We consider the enforcement powers of *program monitors*, which intercept security-sensitive actions of a target application at run time and take remedial steps whenever the target attempts to execute a potentially dangerous action. A common belief in the security community is that program monitors, regardless of the remedial steps available to them when detecting violations, can only enforce safety properties. We formally analyze the properties enforceable by various program monitors and find that although this belief is correct when considering monitors with simple remedial options, it is incorrect for more powerful monitors that can be modeled by *edit automata*. We define an interesting set of properties called *infinite renewal* properties and demonstrate how, when given any reasonable infinite renewal property, to construct an edit automaton that provably enforces that property. We analyze the set of infinite renewal properties and show that it includes every safety property, some liveness properties, and some properties that are neither safety nor liveness.

1 Introduction

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

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 understanding is essential for developing systems that support program monitoring and for developing sound languages for specifying the security policies that these systems can enforce. In addition, well-defined boundaries on the enforcement powers of security 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.

S. De Capitani di Vimercati et al. (Eds.): ESORICS 2005, LNCS 3679, pp. 355–373, 2005.

Schneider defined the first formal models of program monitors and discovered one particularly useful boundary on their power [24]. 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 [18]. When a monitor in this class detects a potential security violation (i.e., "something bad"), it must halt the target.

Although Schneider's result applies only to a particular class of program monitors, other research on formalizing monitors has likewise developed only models that enforce just safety properties. In this paper, we advance the theoretical understanding of practical program monitors by proving that certain types of monitors can enforce non-safety properties. These monitors are modeled by edit automata, which have the power to insert actions on behalf of and suppress actions attempted by the target application. We prove an interesting lower bound on the properties enforceable by such monitors: a lower bound that encompasses strictly more than safety properties.

1.1 Related Work

A rich variety of security monitoring systems has been implemented [14,7,9,11,17,4,8,5]. In general, these systems allow arbitrary code to be executed in response to potential security violations, so they cannot be modeled as monitors that simply halt upon detecting a violation. In most cases, the languages provided by these systems for specifying policies can be considered domain-specific aspect-oriented programming languages [15].

Theoretical efforts to describe security monitoring have lagged behind the implementation work, making it difficult to know exactly which sorts of security policies to expect the implemented systems to be able to enforce. After Schneider made substantial progress by showing that safety properties are an upper bound on the set of policies enforceable by simple monitors [24], Viswanathan, Kim, and others tightened this bound by placing explicit computability constraints on the safety properties being enforced [25,16]. Viswanathan also demonstrated that these computable safety properties are equivalent to coRE properties [25]. Fong then formally showed that placing limits on a monitor's state space induces limits on the properties enforceable by the monitor [12]. Recently, Hamlen, Schneider, and Morrisett compared the enforcement power of static analysis, monitoring, and program rewriting [13]. They showed that the set of statically enforceable properties equals the set of recursively decidable properties of programs, that monitors with access to source-program text can enforce strictly more properties than can be enforced through static analysis, and that program rewriters do not correspond to any complexity class in the arithmetic hierarchy.

In earlier theoretical work, we took a first step toward understanding the enforcement power of monitors that have greater abilities than simply to halt the target when detecting a potential security violation [20]. We introduced *edit automata*, a new model that captures the ability of program monitors to insert actions on behalf of the target and to suppress potentially dangerous actions.

Edit automata are semantically similar to deterministic I/O automata [22] but have very different correctness requirements. The primary contribution of our earlier work was to set up a framework for reasoning about program monitors by providing a formal definition of what it even means for a monitor to enforce a property. Although we also proved the enforcement boundaries of several types of monitors, we did so in a model that assumed that all target programs eventually terminate. Hence, from a practical perspective, our model did not accurately capture the capabilities of real systems. From a theoretical perspective, modeling only terminating targets made it impossible to compare the properties enforceable by edit automata to well-established sets of properties such as safety and liveness properties.

1.2 Contributions

This paper presents the nontrivial generalization of earlier work on edit automata [20] to potentially nonterminating targets. This generalization allows us to reason about the true enforcement powers of an interesting and realistic class of program monitors, and makes it possible to formally and precisely compare this class to previously studied classes.

More specifically, we extend previous work in the following ways.

- We refine and introduce formal definitions needed to understand exactly what it means for program monitors to enforce policies on potentially nonterminating target applications (Section 2). A new notion of enforcement (called *effective$_=$ enforcement*) enables the derivation of elegant lower bounds on the sets of policies monitors can enforce.
- We show why it is commonly believed that program monitors enforce only computable safety properties (Section 3). We show this by revisiting and extending earlier theorems that describe the enforcement powers of simple monitors. The earlier theorems are extended by considering nonterminating targets and by proving that exactly one computable safety property—that which considers everything a security violation—cannot be enforced by program monitors.
- We define an interesting set of properties called *infinite renewal* properties and demonstrate how, when given any reasonable infinite renewal property, to construct an edit automaton that provably enforces that property (Section 4).
- We prove that program monitors modeled by edit automata can enforce strictly more than safety properties. We demonstrate this by analyzing the set of infinite renewal properties and showing that it includes every safety property, some liveness properties, and some properties that are neither safety nor liveness (Section 5).

2 Technical Apparatus

This section provides the formal framework necessary to reason precisely about the scope of policies program monitors can enforce.

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 finite sequences. When τ is a (finite) prefix of (possibly infinite) σ, we write $\tau \preceq \sigma$ or, equivalently, $\sigma \succeq \tau$. If σ has been previously quantified, we often use $\forall \tau \preceq \sigma$ as an abbreviation for $\forall \tau \in \mathcal{A}^\star : \tau \preceq \sigma$; similarly, if τ has already been quantified, 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 key-uniformity policy might be satisfied only by sets of executions such that the cryptographic keys used in all the executions form a uniform distribution over the universe of key values.

Following Schneider [24], 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 because a monitor sees just individual executions and can thus 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* [18,3]. No finite prefix of a valid execution can violate a safety property; stated equivalently: once some finite execution violates the property, all

extensions of that execution violate the property. Formally, $\hat{P}$ is a safety property on a system with action set $\mathcal{A}$ if and only if the following is true.[1]

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

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

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

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

General properties may allow executions to alternate freely between satisfying and violating the property. Such properties are neither safety nor liveness but instead a combination of a single safety and a single liveness property [2]. We show in Section 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 [20]. We model a program monitor formally by a *security automaton* S, which is a deterministic finite or countably infinite state machine (Q, q_0, δ) that is defined with respect to some system with action set $\mathcal{A}$. The set Q specifies the possible automaton states, and q_0 is the initial state. Different automata have slightly different sorts of transition functions (δ), which accounts for the variations in their expressive power. The exact specification of a transition function δ is part of the definition of each kind of security automaton; we only require that δ be complete, deterministic, and Turing Machine computable. We limit our analysis in this work to automata whose transition functions take the current state and input action (the next action the target wants to execute) and return a new state and at most one action to output (make observable). The current input action may or may not be consumed while making a transition.

We specify the execution of each different kind of security automaton S using a labeled operational semantics. The basic single-step judgment has the

[1] Alpern and Schneider [3] model executions as infinite-length sequences of states, where terminating executions contain a final state, infinitely repeated. We can map an execution in their model to one in ours simply by sequencing the events that induce the state transitions (no event induces a repeated final state). With this mapping, it is easy to verify that our definitions of safety and liveness are equivalent to those of Alpern and Schneider.

form $(q, \sigma) \xrightarrow{\tau}_S (q', \sigma')$ where q denotes the current state of the automaton, σ denotes the sequence of actions that the target program wants to execute, q' and σ' denote the state and action sequence after the automaton takes a single step, and τ denotes the sequence of at most one action output by the automaton in this step. The input sequence, σ, is not observable to the outside world whereas the output, τ, is observable.

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

Definition 1 (Multi-step). *The multi-step relation* $(\sigma, q) \overset{\tau}{\Longrightarrow}_S (\sigma', q')$ *is inductively defined as follows (where all metavariables are universally quantified).*

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

In addition, we extend previous work [20] by defining what it means for a program monitor to *transform* a possibly infinite-length input execution into a possibly infinite-length output execution. This definition is essential for understanding the behavior of monitors operating on potentially nonterminating targets.

Definition 2 (Transforms). *A security automaton* $S = (Q, q_0, \delta)$ *on a system with action set* $\mathcal{A}$ transforms *the input sequence* $\sigma \in \mathcal{A}^\infty$ *into the output sequence* $\tau \in \mathcal{A}^\infty$*, notated as* $(q_0, \sigma) \Downarrow_S \tau$*, if and only if the following two constraints are met.*

1. $\forall q' \in Q : \forall \sigma' \in \mathcal{A}^\infty : \forall \tau' \in \mathcal{A}^\star : ((q_0, \sigma) \overset{\tau'}{\Longrightarrow}_S (q', \sigma')) \Rightarrow \tau' \preceq \tau$
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 constraint ensures that S outputs *every* prefix of τ.

2.4 Property Enforcement

Several authors have noted the importance of monitors obeying two abstract principles, which we call *soundness* and *transparency* [19,13,8]. A mechanism that purports to enforce a property $\hat{P}$ is *sound* when it ensures that observable outputs always obey $\hat{P}$; it is *transparent* when it preserves the semantics of executions that already obey $\hat{P}$. We call a sound and transparent mechanism an *effective* enforcer. Because effective enforcers are transparent, they may transform valid input sequences only into semantically equivalent output sequences, for some system-specific definition of semantic equivalence. When two executions $\sigma, \tau \in \mathcal{A}^\infty$ are semantically equivalent, we write $\sigma \cong \tau$. We place no restrictions on a relation of semantic equivalence except that it actually be an equivalence relation (i.e., reflexive, symmetric, and transitive), and that properties should not be able to distinguish between semantically equivalent executions.

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

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

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

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

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

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

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

Definition 4 (Effective$_{=}$ Enforcement). *An automaton S with starting state q_0* effectively$_{=}$ enforces *a property $\hat{P}$ on a system with action set $\mathcal{A}$ if and only if* $\forall \sigma \in \mathcal{A}^\infty : \exists \tau \in \mathcal{A}^\infty :$

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

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

Past research has considered alternative definitions of enforcement [20]. *Conservative* enforcement allows monitors to disobey the transparency requirement, while *precise* enforcement forces effective monitors to obey an additional timing constraint (monitors must accept actions in lockstep with their production by the target). Because these definitions do not directly match the intuitive soundness and transparency requirements of program monitors, we do not study them in this paper.

3 Truncation Automata

This section demonstrates 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. This model is the focus of most of the theoretical work on program monitoring [24,25,16]. Truncation-based monitors have been used successfully to enforce a rich set of interesting safety policies including access control [11], stack inspection [10,1], software fault isolation [26,9], Chinese Wall [6,8,12], and one-out-of-k authorization [12] policies.[2]

Truncation automata have been widely studied, but revisiting them here serves several purposes. It allows us to extend to potentially nonterminating targets previous proofs of their capabilities as effective enforcers [20], to uncover the single computable safety property unenforceable by any sound program monitor, and to provide a precise comparison between the enforcement powers of truncation and edit automata (defined in Section 4).

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

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

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

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

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

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

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

[2] 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.

3.2 Enforceable Properties

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

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

Past research has presented results equating the enforcement power of truncation automata with the set of computable safety properties [25,16,20]. We improve the precision of previous work by showing that there is exactly one computable safety property unenforceable by any sound security automaton: the unsatisfiable safety property, $\forall \sigma \in \mathcal{A}^{\infty} : \neg \hat{P}(\sigma)$. A monitor could never 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 $\hat{P}(\cdot)$. Previous work simply assumed $\hat{P}(\cdot)$ for all $\hat{P}$ [20]; we now show this to be a necessary assumption. The following theorem states that truncation automata effectively$_{=}$ enforce exactly the set of reasonable safety properties.

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

Proof. Please see our companion technical report [21] for the proofs of all the theorems presented in this paper.

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

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

This is an upper bound because a truncation automaton T that effectively$_{\cong}$ enforces $\hat{P}$ must halt at some finite point (having output σ') when its input (σ) violates $\hat{P}$; otherwise, T would accept 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 sequence 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 2 (Effective$_{\cong}$ T^{∞}-Enforcement). *A property $\hat{P}$ on a system with action set $\mathcal{A}$ can be effectively$_{\cong}$ enforced by some truncation automaton T if and only if there exists a decidable predicate D over $\mathcal{A}^{\star}$ such that the following constraints are met.*

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

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.

4 Edit Automata

We now consider the enforcement capabilities of a stronger sort of security automaton called the *edit automaton* [20]. We refine previous work by presenting a more concise formal definition of edit automata. More importantly, we analyze the enforcement powers of edit automata on possibly infinite sequences, which allows us to prove that edit automata can effectively$_{=}$ enforce an interesting, new class of properties that we call *infinite renewal* properties.

4.1 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). We previously defined edit automata that could also perform the following transformations in a single step: insert a finite sequence of actions, accept the current input action, or halt the target [20]. 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.

4.2 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 [23]. Monitoring systems like virtual machines can also be used in this way, feigning execution of a sequence of the target's actions and only making the sequence observable when it is known to be valid.

As we did for truncation automata, we develop a lower bound on the set of properties that edit automata effectively$_{\cong}$ enforce by considering the properties they effectively$_{=}$ enforce. Using the above-described technique of suppressing invalid inputs until the monitor determines that the suppressed input obeys a property, edit automata can effectively$_{=}$ enforce any reasonable *infinite renewal* (or simply *renewal*) property. A renewal property is one in which every valid infinite-length sequence has infinitely many valid prefixes, and conversely, every invalid infinite-length sequence has only finitely many valid prefixes. For example, a property $\hat{P}$ may be satisfied only by executions that contain the action a.

This is a renewal property because valid infinite-length executions contain an infinite number of valid prefixes (in which a appears) while invalid infinite-length executions contain only a finite number of valid prefixes (in fact, zero). This $\hat{P}$ is also a liveness property because any invalid finite execution can be made valid simply by appending the action a. Although edit automata cannot enforce this $\hat{P}$ because $\neg\hat{P}(\cdot)$, in Section 5.2 we will recast this example as a reasonable "eventually audits" policy and show several more detailed examples of renewal properties enforceable by edit automata.

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

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

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

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

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 (possibly invalid) output into a valid one.

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 in the companion technical report [21].

Theorem 4 (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. $\forall \sigma \in \mathcal{A}^\omega : \hat{P}(\sigma) \iff \left(\begin{array}{l} \forall \sigma' \preceq \sigma : \exists \tau \preceq \sigma : \sigma' \preceq \tau \wedge \hat{P}(\tau) \\ \vee\ \hat{P}(\sigma) \wedge \\ \quad \exists \sigma' \preceq \sigma : \forall \tau \succeq \sigma' : \hat{P}(\tau) \Rightarrow \tau = \sigma \wedge \\ \quad \textit{the existence and actions of } \sigma \\ \quad\ \textit{are computable from } \sigma' \end{array} \right)$
2. $\hat{P}(\cdot)$
3. $\forall \sigma \in \mathcal{A}^\star : \hat{P}(\sigma)$ *is decidable*

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

Theorem 5 (Effective$_\cong$ E^∞-Enforcement). *Let D be a decidable function $D : \mathcal{A}^\star \times \mathcal{A}^\star \to \mathcal{A} \cup \{\cdot\}$. Then $R^\star_D$ is a decidable function $R^\star_D : \mathcal{A}^\star \times \mathcal{A}^\star \times \mathcal{A}^\star \to \mathcal{A}^\star$ parameterized by D and inductively defined as follows, where all metavariables are universally quantified.*

- $R^\star_D(\cdot, \sigma, \tau) = \tau$
- $(D(\sigma; a, \tau) = \cdot) \Rightarrow R^\star_D(a; \sigma', \sigma, \tau') = R^\star_D(\sigma', \sigma; a, \tau')$
- $(D(\sigma; a, \tau) = a') \Rightarrow R^\star_D(a; \sigma', \sigma, \tau') = R^\star_D(a; \sigma', \sigma, \tau'; a')$

A property $\hat{P}$ on a system with action set $\mathcal{A}$ can be effectively$_\cong$ enforced by some edit automaton E if and only if there exists a decidable D 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. $\forall \sigma' \preceq \sigma : \forall \tau' \in \mathcal{A}^\star : (R^\star_D(\sigma', \cdot, \cdot) = \tau') \Rightarrow \tau' \preceq \tau$
2. $\forall \tau' \preceq \tau : \exists \sigma' \preceq \sigma : R^\star_D(\sigma', \cdot, \cdot) = \tau'$

3. $\hat{P}(\tau)$
4. $\hat{P}(\sigma) \Rightarrow \sigma \cong \tau$

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

Limitations. In addition to standard assumptions of program monitors, such as that a target cannot circumvent or corrupt a monitor, our theoretical model makes assumptions particularly relevant to edit automata that are sometimes violated in practice. Most importantly, our model assumes that security automata have the same computational capabilities as the system that observes the monitor's output. If an action violates this assumption by requiring an outside system in order to be executed, it cannot be "feigned" (i.e., suppressed) by the monitor. For example, it would be impossible for a monitor to feign sending email, wait for the target to receive a response to the email, test whether the target does something invalid with the response, and then decide to "undo" sending email in the first place. Here, the action for sending email has to be made observable to systems outside of the monitor's control in order to be executed, so this is an unsuppressible action. A similar limitation arises with time-dependent actions, where an action cannot be feigned (i.e., suppressed) because it may behave differently if made observable later. In addition to these sorts of unsuppressible actions, a system may contain actions uninsertable by monitors because, for example, the monitors lack access to secret keys that must be passed as parameters to the actions. In the future, we plan to explore the usefulness of including sets of unsuppressible and uninsertable actions in the specification of systems. We might be able to harness earlier work [20], which defined security automata limited to inserting (insertion automata) or suppressing (suppression automata) actions, toward this goal.

5 Infinite Renewal Properties

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

5.1 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. The primary result of the current work, that edit automata can enforce any reasonable renewal property, agrees with the finding in earlier work that edit automata can enforce *every*

reasonable property on systems that only exhibit finite executions [20]. Without infinite-length executions, every property is a renewal property.

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

There are renewal properties that are not liveness properties (e.g., the property that is only satisfied by the empty sequence), and there are liveness properties that are not renewal properties (e.g., the nontermination property only satisfied by infinite executions). Some renewal properties, such as the one only satisfied by the empty sequence and the sequence $a;a$, are neither safety nor liveness. Although Alpern and Schneider [3] 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.

5.2 Example Properties

We next present several examples of renewal properties that are not safety properties, as well as some examples of non-renewal properties. By the theorems in Sections 3.2 and 4.2, the non-safety renewal properties are effectively$_=$ enforceable by edit automata but not by truncation automata. Moreover, the proof of Theorem 3 in our companion technical report [21] shows how to construct an edit automaton to enforce any of the renewal properties described in this subsection.

Renewal properties. Suppose we wish to constrain a user's interaction with a computer system. A user may first obtain credentials (e.g., a Kerberos ticket) and then log in. If he has obtained no credentials then executing a log-in action causes him to be logged in as a guest. At no time, however, can the user log in as "root." The process of logging in to the system may repeat indefinitely, so we might write the requisite property $\hat{P}$ more specifically as $(a_1{}^\star; a_2)^\infty$, where a_1 ranges over all actions for obtaining credentials, a_2 over actions for logging in, and a_3 over actions for logging in as root.[3] 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 non-safety, non-liveness property is a renewal property because infinite-length executions are valid if and only if they contain infinitely many (valid) prefixes of the form $(a_1{}^\star; a_2)^\star$.

[3] As noted by Alpern and Schneider [3], this sort of $\hat{P}$ might be expressed with the (strong) *until* operator in temporal logic; event a_1 occurs *until* event a_2.

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 4.2. 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 4.2, edit automata derive their power from being able to operate in a way similar to intentions files in database transactions. At a high-level, any transaction-based property is a renewal property. Let τ range over finite sequences of single, valid transactions. A transaction based policy could then be written as τ^∞; a valid execution is one containing any number of valid transactions. Such transactional properties can be non-safety because executions may be invalid within a transaction but become valid at the conclusion of that transaction. Transactional properties can also be non-liveness when there exists a way to irremediably corrupt a transaction (e.g., every transaction beginning with *start*;*self-destruct* is illegal). Nonetheless, transactional properties are renewal properties because infinite-length executions are valid if and only if they contain an infinite number of prefixes that are valid sequences of transactions. The renewal properties described above as matching sequences of the form $(a_1{}^\star; a_2)^\infty$ can also be viewed as transactional; each transaction must match $a_1{}^\star; a_2$.

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

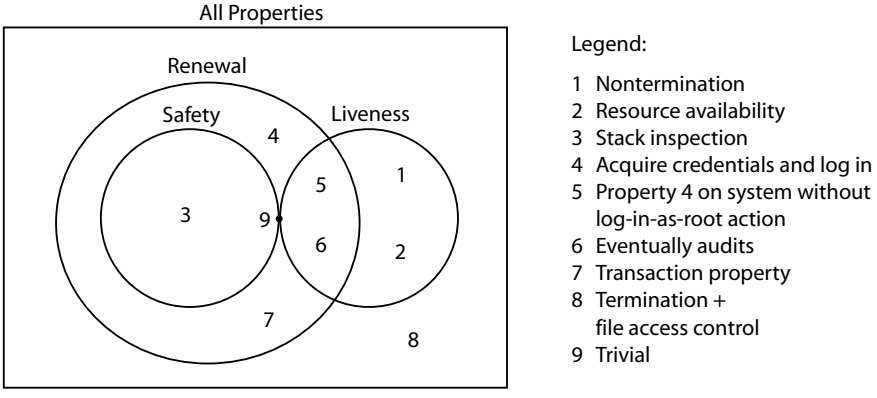

Fig. 1. Relationships between safety, liveness, and renewal properties

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

Figure 1 summarizes the results of the preceding discussion and that of Section 5.1. The Trivial property considers all executions legal and is the only property in the intersection of safety and liveness properties.

6 Conclusions

When considering the space of security properties enforceable by monitoring potentially nonterminating targets, we have found that a simple variety of monitor enforces exactly the set of computable and satisfiable safety properties while a more powerful variety can enforce any computable infinite renewal property that is satisfied by the empty sequence. Because our model permits infinite sequences of actions, it is compatible with previous research on safety and liveness properties.

Awareness of formally proven bounds on the power of security mechanisms facilitates our understanding of policies themselves and the mechanisms we need to enforce them. For example, observing that a stack-inspection policy is really just an access-control property (where access is granted or denied based on the history of function calls and returns), which in turn is clearly a safety property, makes it immediately obvious that simple monitors modeled by truncation automata are sufficient for enforcing stack-inspection policies. Similarly, if we can

observe that infinite executions in a property specifying how users log in are valid if and only if they contain infinitely many valid prefixes, then we immediately know that monitors based on edit automata can enforce this renewal property. We hope that with continued research into the formal enforcement bounds of various security mechanisms, security architects will be able to pull from their enforcement "toolbox" exactly the right sorts of mechanisms needed to enforce the policies at hand.

Acknowledgments

We wish to thank Ed Felten for pointing out the operational similarity between edit automata and database intentions files. In addition, Kevin Hamlen, Fred Schneider, Greg Morrisett, and the anonymous reviewers provided insightful comments on an earlier version of this paper. ARDA grant NBCHC030106, DARPA award F30602-99-1-0519, and NSF grants CCR-0238328 and CCR-0306313 have provided support for this research.

References

1. M. Abadi and C. Fournet. Access control based on execution history. In *Proceedings of the 10th Annual Network and Distributed System Symposium*, Feb. 2003.
2. B. Alpern and F. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1987.
3. B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, Oct. 1985.
4. L. Bauer, J. Ligatti, and D. Walker. Types and effects for non-interfering program monitors. In M. Okada, B. Pierce, A. Scedrov, H. Tokuda, and A. Yonezawa, editors, *Software Security—Theories and Systems. Mext-NSF-JSPS International Symposium, ISSS 2002, Tokyo, Japan, November 8-10, 2002, Revised Papers*, volume 2609 of *Lecture Notes in Computer Science*. Springer, 2003.
5. L. Bauer, J. Ligatti, and D. Walker. Composing security policies with polymer. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, Chicago, June 2005.
6. D. F. C. Brewer and M. J. Nash. The chinese wall security policy. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 206–214, 1989.
7. G. Edjlali, A. Acharya, and V. Chaudhary. History-based access control for mobile code. In *ACM Conference on Computer and Communications Security*, pages 38–48, 1998.
8. Ú. Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Cornell University, Jan. 2004.
9. Ú. Erlingsson and F. B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the New Security Paradigms Workshop*, pages 87–95, Caledon Hills, Canada, Sept. 1999.
10. Ú. Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 246–255, Oakland, California, May 2000.

11. D. Evans and A. Twyman. Flexible policy-directed code safety. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1999.
12. P. W. L. Fong. Access control by tracking shallow execution history. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, California, USA, May 2004.
13. K. Hamlen, G. Morrisett, and F. Schneider. Computability classes for enforcement mechanisms. Technical Report TR2003-1908, Cornell University, Aug. 2003.
14. C. Jeffery, W. Zhou, K. Templer, and M. Brazell. A lightweight architecture for program execution monitoring. In *PASTE '98: Proceedings of the 1998 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 67–74. ACM Press, 1998.
15. G. Kiczales, J. Irwin, J. Lamping, J.-M. Loingtier, C. V. Lopes, C. Maeda, and A. Mendhekar. Aspect-oriented programming. *ACM Comput. Surv.*, 28(4es):154, 1996.
16. M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswantathan. Computational analysis of run-time monitoring—fundamentals of Java-MaC. In *Run-time Verification*, June 2002.
17. M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky. Formally specified monitoring of temporal properties. In *European Conference on Real-time Systems*, York, UK, June 1999.
18. L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions of Software Engineering*, 3(2):125–143, 1977.
19. J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. Technical Report TR-681-03, Princeton University, May 2003.
20. J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1–2):2–16, Feb. 2005.
21. J. Ligatti, L. Bauer, and D. Walker. Enforcing non-safety security policies with program monitors. Technical Report TR-720-05, Princeton University, Jan. 2005.
22. N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the 6th annual ACM Symposium on Principles of distributed computing*, pages 137–151. ACM Press, 1987.
23. W. H. Paxton. A client-based transaction system to maintain data integrity. In *Proceedings of the 7th ACM symposium on Operating systems principles*, pages 18–23. ACM Press, 1979.
24. F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and Systems Security*, 3(1):30–50, Feb. 2000.
25. M. Viswanathan. *Foundations for the Run-time Analysis of Software Systems*. PhD thesis, University of Pennsylvania, 2000.
26. R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 203–216, Asheville, Dec. 1993.