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 to construct a program monitor that provably enforces any reasonable infinite renewal property. 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

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 those 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 on purely run-time program monitors has likewise only focused on their ability 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.

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 2004; 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). Researchers have devised many techniques for proving that programs obey such automata-specified safety properties [Walker 2000; Hamlen et al. 2006b; Aktug et al. 2008].

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 either *accept* the actions of untrusted targets or *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).

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 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-

[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.

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. 2006a]. 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 user-level operating-system extensions (e.g., user-level file systems) 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 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 contain some nonsafety policies and are a superset of the monitor-enforceable policies, and, interestingly, they prove that the RW-enforceable policies do not correspond to *any* complexity class in the arithmetic hierarchy.

Security Automata in Process Algebras. As part of the S3MS (Security of Software and Services for Mobile Systems) project, Martinelli and Matteucci have modeled security automata (including truncation and edit automata) as operators in process algebras [Martinelli and Matteucci 2007b; 2007a], particularly in variants of Milner's Calculus of Communicating Systems (*CCS*) [Milner 1978]. Having modeled security automata and their semantics, Martinelli and Matteucci demonstrate techniques for synthesizing security automata from policies specified as formulae in temporal logic. Matteucci has also extended the framework to accommodate reasoning about time (using a timed variant of *CCS*) [Matteucci 2007], and Matteucci, Martinelli, and Mori recently implemented their monitor-synthesis designs in practical tools [Matteucci 2006; Martinelli and Mori 2007].

1.2 Contributions

We extend previous work in three 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 *reasonable 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 can sometimes enforce nonsafety properties.

This article draws heavily from our earlier conference paper "Enforcing Nonsafety Security Policies with Program Monitors" [Ligatti et al. 2005b] but extends that earlier work in many ways.

—We have given related work a much more complete treatment in Section 1.1 and have added material to our description of ongoing and future work in Section 6.2.

—We include proofs for theorems. Most importantly, the proof of Theorem 3.3 in Appendix A shows how, when given any reasonable infinite renewal property, to construct a program monitor that provably enforces that property.

—Although we include three theorems stated in the earlier conference paper (Theorems 3.1, 3.2, and 3.3), we have added two new theorems (Theorems 2.5 and 3.8) and have formalized another (Theorem 3.4) that was only sketched in earlier work. The nontrivial formalization of Theorem 3.4 and the new Theorems 2.5 and 3.8 shed more light on the exact set of properties edit automata (and security automata in general) can enforce.

—In the new Section 5, we discuss several limitations and practical considerations relevant to our enforcement model. We also briefly describe Polymer [Bauer et al. 2005a; Ligatti 2006], an edit-automaton-inspired, implemented enforcement mechanism that can enforce some nonsafety policies.

—Finally, we make numerous minor corrections and additions to the original material. For instance, Section 2.1 includes a correction to our notation for sequence concatenation (the original notation only applied when concatenating two finite sequences, but we often need this notation to denote the concatenation 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 to arrive at a precise definition of what it means for a monitor to enforce a policy.

By necessity, our models abstract from and idealize actual systems. Section 5 discusses some of the implications and limitations of this idealization.

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*, or trace, is simply a finite or infinite sequence of actions; a finite sequence indicates a terminating execution, while an infinite sequence indicates a nonterminating execution. The set of all finite executions on a system with action set $\mathcal{A}$ is denoted 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, $\sigma\succeq\tau$. When τ is a *strict* prefix of σ (i.e., $\tau\preceq\sigma$ and $\tau\neq\sigma$), we write $\tau\prec\sigma$ or $\sigma\succ\tau$.

Given some σ, we often use $\forall\tau\preceq\sigma$ as an abbreviation for $\forall\tau \in \mathcal{A}^\star : \tau\preceq\sigma$ and $\exists\tau\preceq\sigma$ for $\exists\tau \in \mathcal{A}^\star : \tau\preceq\sigma$. Similarly, when given some τ, we use $\forall\sigma\succeq\tau$ as an abbreviation for $\forall\sigma \in \mathcal{A}^\infty : \sigma\succeq\tau$ and $\exists\sigma\succeq\tau$ for $\exists\tau \in \mathcal{A}^\infty : \sigma\succeq\tau$. We also use analogous abbreviations with $\prec$ in place of $\preceq$ and $\succ$ in place of $\succeq$.

2.2 Policies and Properties

A *security policy* is a predicate P on sets of (finite- or infinite-length) 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 decidable *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*, *legal*, or *good*. Likewise, when $\neg\hat{P}(\tau)$, we say that τ *violates* or *disobeys* the property, or that τ is *invalid*, *illegal*, or *bad*.

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 execution can always be extended to satisfy 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. Alpern and Schneider showed that such properties are neither safety nor liveness but instead the conjunction of a single safety and a single liveness property [Alpern and Schneider 1985; 1987; Schneider 1987]. The decomposition can be straightforward: given a property $\hat{P}$, define property $\hat{P}_S$ to

[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 building a sequence of the actions that induce the state transitions in the execution in their model (no event induces a repeated final state). For example, consider the Alpern and Schneider execution $\mathtt{s}_1; \mathtt{s}_2; \mathtt{s}_1; \mathtt{s}_1; \mathtt{s}_1; \ldots$ and assume that action $\mathtt{a}_1$ induces the transition from $\mathtt{s}_1$ to $\mathtt{s}_2$ and $\mathtt{a}_2$ induces the transition from $\mathtt{s}_2$ to $\mathtt{s}_1$. In our model, this execution would be $\mathtt{a}_1; \mathtt{a}_2$. With this mapping, it is easy to verify that our definitions of safety and liveness are equivalent to those of Alpern and Schneider.

be satisfied exactly by those executions that satisfy $\hat{P}$ or that have finite length and can be extended to satisfy $\hat{P}$; also define property $\hat{P}_L$ to be satisfied exactly by those executions that satisfy $\hat{P}$ or that have finite length and cannot be extended to satisfy $\hat{P}$. It is not difficult to show that $\hat{P}_S$ is a safety property, $\hat{P}_L$ is a liveness property, and that an execution satisfies $\hat{P}$ if and only if it satisfies both $\hat{P}_S$ and $\hat{P}_L$. Section 4.1 contains a simple example of this decomposition technique.

Finally, in our analyses we will often find it convenient to consider only a subset of properties that we call the *reasonable* properties.

Definition 2.1 *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

2.3 Security Automata

Program monitors operate by *transforming* execution sequences of an untrusted target application at run time to ensure that all observable executions satisfy some property. We model a program monitor formally by a *security automaton* S, which is a deterministic, finite or countably infinite state machine (Q, q_0, δ) that is defined with respect to some system with action set $\mathcal{A}$. The set Q specifies the possible automaton states, and q_0 is the initial state. Different automata have slightly different sorts of transition functions (δ), which accounts for the variations in their expressive power. The exact specification of a transition function δ is part of the definition of each kind of security automaton; we only require that δ be total, 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 and ready to execute). A transition of the automaton is triggered by the presence of an input action. The current input action may or may not be consumed while making a transition; this is different from some previous models, which required that an input action always be consumed during a transition [Schneider 2000].

We specify the execution of each 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 2.2 *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 2.3 *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 $\tau \in \mathcal{A}^\infty$, denoted as $(q_0, \sigma) \Downarrow_S \tau$, if and only if the following 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.

Section 5.3 also discusses the possibility of monitors transforming input executions from a more limited domain than $\mathcal{A}^\infty$. This possibility may in some cases give monitors more power, but it also raises a host of complex issues.

2.4 Property Enforcement

We and several others concurrently noted the importance of monitors obeying two abstract principles, which we call *soundness* and *transparency* [Ligatti et al. 2003; Erlingsson 2004; Hamlen et al. 2006a]. A mechanism that enforces 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 semantic-equivalence relation except that it actually be an equivalence relation (i.e., reflexive, symmetric, and transitive), and that properties of interest 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 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 2.4 *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) $\hat{P}(\sigma) \implies \sigma \approx \tau$

Although semantic equivalence allows us to model ideal enforcement behavior, and captures some realistic enforcement behaviors that would otherwise be neglected, we have found (and this article shows) that in many situations the system-specific equivalence relation $\approx$ complicates our theorems and proofs without making them significantly more enlightening. A major difficulty with semantic equivalence is its generality: for *any* reasonable property $\hat{P}$ there exists a sufficiently helpful equivalence relation that enables a security automaton to enforce $\hat{P}$.

THEOREM 2.5 PERMISSIBILITY OF $\approx$. *For all reasonable properties $\hat{P}$ on a system with action set $\mathcal{A}$, there exists an equivalence relation $\approx$ and security automaton S such that S effectively$_{\approx}$ enforces $\hat{P}$.*

PROOF. Define the equivalence relation $\approx$ such that $\forall \sigma, \sigma' \in \mathcal{A}^\infty : \sigma \approx \sigma' \iff (\hat{P}(\sigma) \wedge \hat{P}(\sigma'))$. Define S to have a single state q and never make any state transitions or output any actions. Then, $\forall \tau \in \mathcal{A}^\infty : (q, \tau) \Downarrow_S \cdot$. By assumption, $\hat{P}(\cdot)$; hence, S is sound. All valid executions are equivalent to $\cdot$, so S is also transparent. Therefore, S effectively$_{\approx}$ enforces $\hat{P}$. □

Because semantic equivalence can be such a powerful aid in enforcement, we have found that we can sometimes gain more insight into the enforcement powers of program monitors by limiting our analyses to systems in which the equivalence relation ($\approx$) is just syntactic equality ($=$). We call effective$_{\approx}$ enforcers operating on such systems *effective$_{=}$ enforcers.* To formalize 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 for which the system-specific equivalence relation ($\approx$) is the system-unspecific equality relation ($=$).

Definition 2.6 *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, in our model, 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. MONITOR-ENFORCEABLE POLICIES

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 widely studied variety that we model with *truncation automata* (Section 3.1) and a more complex 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 truncating 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 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 2004; 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 deterministic and total function $\delta : Q \times \mathcal{A} \rightarrow Q \cup \{\mathit{halt}\}$ 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*). To ensure that the transition function is deterministic, we require that $\mathit{halt} \notin Q$. The following rules formally specify the operational semantics of truncation automata.

[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.

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

$$\frac{\sigma = a;\sigma' \quad \delta(q,a) = q'}{(q,\sigma) \xrightarrow{a}_T (q',\sigma')} \quad (\text{T-}\textsc{Step}) \qquad \frac{\sigma = a;\sigma' \quad \delta(q,a) = halt}{(q,\sigma) \xrightarrow{\cdot}_T (q,\cdot)} \quad (\text{T-}\textsc{Stop})$$

As described in Section 2.3, we extend this 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]. Here we show that there is exactly one computable safety property unenforceable by any sound truncation automaton: the unsatisfiable safety property that considers all executions invalid. We show this by proving that no security automaton can enforce a property that considers empty executions valid. The only safety property for which $\neg\hat{P}(\cdot)$ holds is the unsatisfiable policy; for truncation automata this implies that the unsatisfiable property cannot be enforced. That security automata cannot enforce unsatisfiable properties was independently (and concurrently with our original derivation [Ligatti et al. 2005b]) shown by Hamlen, Morrisett, and Schneider [Hamlen et al. 2006a]. However, we show more generally that security automata also cannot enforce satisfiable properties $\hat{P}$ such that $\neg\hat{P}(\cdot)$.

A monitor in our framework cannot enforce such a property because there is no valid output sequence that can be produced in response to an empty, invalid input sequence. To prevent this case and to ensure that security automata can behave correctly on targets that generate no actions, we require that the empty sequence satisfies any property we are interested in enforcing. Previous work assumed for convenience that $\hat{P}(\cdot)$ [Bauer et al. 2002], but here we prove that it is a necessity.

The following theorem states that truncation automata effectively$_{=}$ enforce exactly the set of reasonable safety properties. Although previous work has proved similar results [Schneider 2000; Kim et al. 2002; Viswanathan 2000; Ligatti et al. 2005a], we include the theorem and proof in this article in order to transfer the previous results to our model of monitors as execution transformers.

Theorem 3.1 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*) $\forall\sigma \in \mathcal{A}^\infty : \neg\hat{P}(\sigma) \implies \exists\sigma'\preceq\sigma : \forall\tau\succeq\sigma' : \neg\hat{P}(\tau)$ (Safety)

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

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

Proof. Please see Appendix A for the proofs of all theorems presented in this section. □

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 3.2 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*) $\neg D(\cdot)$

Hence, truncation automata can effectively$_\approx$ enforce properties not effectively$_=$ enforceable (i.e., nonsafety properties) only when the property and the system's relation of semantic equivalence allow some invalid execution σ to have a valid prefix σ' that (1) can be extended to a valid execution τ such that $\sigma\preceq\tau$, and (2) is equivalent to *all* valid extensions of σ'.

One can imagine simple examples that satisfy these constraints. Consider, for instance, a system with actions `on` and `off`. On this system, any subexecution of the form $(\texttt{on};\texttt{off})^\star$ is equivalent to the empty sequence. The property $\hat{P}$ we wish to enforce requires that only executions of the form $(\texttt{on};\texttt{off})^\star$ are valid (e.g., $\neg\hat{P}((\texttt{on};\texttt{off})^\omega)$ and $\neg\hat{P}(\texttt{on})$). This $\hat{P}$ is a nonsafety property because $\neg\hat{P}(\texttt{on})$ but $\hat{P}(\texttt{on};\texttt{off})$, and so by Theorem 3.1 it cannot be effectively$_=$ enforced by a truncation automation. On the other hand, following the transitions specified in the proof of Theorem 3.2 in Appendix A, a truncation automaton can effectively$_\approx$ enforce $\hat{P}$ by immediately halting before accepting any actions.

On practical systems, it seems unlikely that a property requiring enforcement and a system's relation of semantic equivalence would be so broadly defined. We therefore consider the set of properties detailed in Theorem 3.1 (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 deterministic and total transition function δ of an edit automaton has the form $\delta : Q \times \mathcal{A} \rightarrow Q \times (\mathcal{A} \cup \{\cdot\})$. 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.

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

$$\frac{\sigma = a;\sigma' \quad \delta(q, a) = (q', a')}{(q, \sigma) \xrightarrow{a'}_E (q', \sigma)} \quad \text{(E-INS)} \qquad \frac{\sigma = a;\sigma' \quad \delta(q, a) = (q', \cdot)}{(q, \sigma) \xrightarrow{\cdot}_E (q', \sigma')} \quad \text{(E-SUP)}$$

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 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. We discuss some practical considerations and limitations of this model, such as the difficulty of ensuring that suppressed sequences of actions get inserted atomically, in Section 5.

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.

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

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

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

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 3.3 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 eagerly 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. Intuitively, constraint (1) in the theorem is a weak version of Renewal$_2$; it permits a valid infinite-length execution σ to have a prefix σ' that an eager-insertion function f can decide to extend, action by action, to σ. Constraint (2) restricts the behavior of f to ensure that it only specifies a sequence of eagerly insertable actions in cases where an automaton, after inserting those actions, will be guaranteed that its output is a prefix of (or equal to) any valid extension of the current input. In general, effective$_=$ enforcers can eagerly insert a sequence of actions σ if and only if all valid extensions of the current input σ' begin by extending σ' with σ. Finally, constraints (3) and (4) limit consideration to reasonable properties.

Theorem 3.4 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 there exists decidable* eager-insertion function $f : (\mathcal{A}^\star \times \mathbb{N}) \rightarrow (\{\cdot\} \cup \mathcal{A})$ *such that the following constraints are met.*

(1) $\forall \sigma \in \mathcal{A}^\omega : \hat{P}(\sigma) \iff \begin{pmatrix} \forall \sigma' \preceq \sigma : \exists \tau \preceq \sigma : \sigma' \preceq \tau \wedge \hat{P}(\tau) \\ \vee\ \exists \sigma' \preceq \sigma : \sigma = \sigma'; f(\sigma',0); f(\sigma',1); f(\sigma',2); \ldots \end{pmatrix}$

(2) $\forall \sigma \in \mathcal{A}^\infty : \forall \sigma' \in \mathcal{A}^\star : (f(\sigma',0) \neq \cdot \wedge \sigma = \sigma'; f(\sigma',0); f(\sigma',1); f(\sigma',2); \ldots) \implies$

(a) $\hat{P}(\sigma)$

(b) $\sigma \in \mathcal{A}^\star \implies (\forall \tau \succeq \sigma' : \hat{P}(\tau) \implies \sigma \preceq \tau)$

(c) $\sigma \in \mathcal{A}^\omega \implies (\forall \tau \succeq \sigma' : \hat{P}(\tau) \implies \sigma = \tau)$

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

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

Theorems 3.3 and 3.4 show that edit automata exercise their full effective$_=$ enforcement powers by operating in a transactional manner. The automaton may complete transactions:

—lazily, by suppressing a finite sequence of dangerous actions σ and later inserting σ when it is known to be valid; or

—eagerly, by inserting the unique sequence of actions σ (which may have infinite length) that completes the current transaction and later suppressing σ if it has finite length and actually does appear in the input.

Turning our attention to effective$_{\approx}$ enforcement, we will find that edit automata also exercise their full effective$_{\approx}$ enforcement powers in a similar transactional manner. However, in the case of effective$_{\approx}$ enforcement, our analysis is more complex and relies on several auxiliary definitions (Definitions 3.5, 3.6, and 3.7) to encapsulate the ways edit automata may output action-sequence responses to valid input transactions. The casual reader may wish to skip the remainder of this section and accept in summary that edit automata as fully powerful effective$_{=}$ and effective$_{\approx}$ enforcers monitor input transactions and respond lazily or eagerly (as itemized above). In the case of effective$_{\approx}$ enforcement, though, the analysis becomes complex because we must constrain many sequences to be equivalent rather than just assuming equality and, conversely, can sometimes only assume equivalence where equality would be simpler.

Our first auxiliary definition defines a type of function that can return the next action to output in response to (or in anticipation of) a valid input transaction. The definition uses some new notation. Let $\mathcal{A}^+$ be the set of all nonempty, finite sequences of actions on a system with action set $\mathcal{A}$. Then, an element of $(\mathcal{A}^+)^\star$ is a possibly empty, finite sequence of executions $(\sigma_0, .., \sigma_n)$ such that every execution in the sequence is an element of $\mathcal{A}^+$.

Definition 3.5 *Transaction Action-output Function.* A function f is a *transaction action-output function* if and only if:

(1) f has the type $f : ((\mathcal{A}^+)^\star \times \mathbb{N}) \rightarrow (\mathcal{A} \cup \{\cdot\})$

(2) f is decidable over all inputs $((\sigma_0, .., \sigma_n), m)$ such that for all i $(0 \leq i < n)$ there exists a $j \in \mathbb{N}$ such that $f((\sigma_0, .., \sigma_i), j) = \cdot$

(3) $f(\cdot, 0) = \cdot$

(4) $\forall S \in (\mathcal{A}^+)^\star : \forall m \in \mathbb{N} : (f(S, m) = \cdot \implies \forall n{>}m : f(S, n) = \cdot)$

The first argument to a transaction action-output function is, roughly speaking, a history of valid transactions in an input execution. The second argument, n, indicates that the function should return the n^{th} action (starting at 0) that the monitor outputs in response to the most recently completed input transaction (or $\cdot$ if no such action exists). Constraint (2) requires the function to be decidable whenever all of its earlier responses (i.e., sequence outputs) to valid input transactions are finite; it does not make sense to ask for a response from the function when it previously responded by outputting an infinite-length sequence, and our analysis will exclude or consider irrelevant such a possibility. Constraint (3) prohibits a transaction action-output function from considering the empty execution a valid transaction, and constraint (4) ensures that a transaction action-output function cannot complete a response to a valid transaction and then later decide to output additional actions in response to the same transaction.

Definition 3.6 straightforwardly generalizes transaction action-output functions to full-transaction output functions. A full-transaction output function returns the complete (possibly infinite-length) sequence of actions to output in response to a valid input transaction.

Definition 3.6 *Full-transaction Output Function.* Given any transaction action-output function f, define the *full-transaction output function* $F_f : (\mathcal{A}^+)^\star \rightarrow \mathcal{A}^\infty$ as follows: $F_f(\sigma_0, \sigma_1, .., \sigma_n) = f((\sigma_0, .., \sigma_n), 0); f((\sigma_0, .., \sigma_n), 1); f((\sigma_0, .., \sigma_n), 2); \ldots$

Similarly, Definition 3.7 generalizes full-transaction output functions to full output functions. A full output function returns the complete sequence of actions to output in response to an entire sequence of valid input transactions (i.e., an entire input execution).

Definition 3.7 *Full Output Function.* Given any full-transaction output function F_f, define the *full output function* $O_f : (\mathcal{A}^+)^\infty \rightarrow \mathcal{A}^\infty$ as follows.

(1) $\forall(\sigma_0, .., \sigma_n) \in (\mathcal{A}^+)^\star : O_f(\sigma_0, .., \sigma_n) = F_f(\sigma_0); F_f(\sigma_0, \sigma_1); ..; F_f(\sigma_0, .., \sigma_n)$
(2) $\forall(\sigma_0, \sigma_1, \ldots) \in (\mathcal{A}^+)^\omega : O_f(\sigma_0, \sigma_1, \ldots) = F_f(\sigma_0); F_f(\sigma_0, \sigma_1); \ldots$

With Definitions 3.5, 3.6, and 3.7 in hand, we delineate the properties effectively$_\approx$ enforceable by edit automata in Theorem 3.8, below. At a high level, Theorem 3.8 states that edit automata effectively$_\approx$ enforce exactly those properties that can be enforced by outputting (possibly infinite-length) responses to input transactions, subject to the standard enforcement restriction that all overall responses must be valid and equivalent to valid inputs. More specifically, the two constraints in Theorem 3.8 consider complementary scenarios: (1) considers input executions that have a finite number of transactions but allows the monitor to respond to the final (possibly invalid) input transaction with an infinite-length output; (2) considers input executions that have an infinite number of valid transactions, each prompting a finite response from the monitor. Constraints (1a), (1b), (2a), and (2b) place straightforward restrictions on the overall sequences an effective$_\approx$ enforcer can output in response to the sequences of input transactions: the overall output must be valid and equivalent to any valid input. Finally, constraints (1c) and (1d) handle special cases in which an edit automaton processes a finite number of transactions, but in doing so fails to respond to all of its valid input. Specifically, (1c) handles the possibility of outputting nothing in response to the tail end of a valid input, while (1d) handles the possibility of outputting an infinite sequence of actions in response to a strict prefix of a valid input.

Theorem 3.8 Effective$_\approx$ E^∞-Enforcement. *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* transaction action-output function f *such that the following constraints are met.*

(1) $\forall(\sigma_0, .., \sigma_n) \in (\mathcal{A}^+)^\star : (n < 1 \ \vee \ O_f(\sigma_0, .., \sigma_{n-1}) \in \mathcal{A}^\star) \implies$
  (a) $\hat{P}(O_f(\sigma_0, .., \sigma_n))$
  (b) $\hat{P}(\sigma_0; ..; \sigma_n) \implies \sigma_0; ..; \sigma_n \approx O_f(\sigma_0, .., \sigma_n)$
  (c) $\forall \tau \succ (\sigma_0; ..; \sigma_n) : (\hat{P}(\tau) \ \wedge \ O_f(\tau){=}O_f(\sigma_0, .., \sigma_n)) \implies \tau \approx O_f(\sigma_0, .., \sigma_n)$
  (d) $\forall \tau \in \mathcal{A}^\infty : (\hat{P}(\tau) \ \wedge \ \sigma_0; ..; \sigma_n \prec \tau \ \wedge \ O_f(\sigma_0; ..; \sigma_n) \in \mathcal{A}^\omega) \implies \tau \approx O_f(\sigma_0, .., \sigma_n)$
(2) $\forall(\sigma_0, \sigma_1, \ldots) \in (\mathcal{A}^+)^\omega : (\forall i \in \mathbb{N} : O_f(\sigma_0, .., \sigma_i) \in \mathcal{A}^\star) \implies$
  (a) $\hat{P}(O_f(\sigma_0, \sigma_1, \ldots))$
  (b) $\hat{P}(\sigma_0; \sigma_1; \ldots) \implies (\sigma_0; \sigma_1; \ldots) \approx O_f(\sigma_0, \sigma_1, \ldots)$

Some properties can be effectively$_{\approx}$ but not effectively$_{=}$ enforced by edit automata. For instance, recall the property $\hat{P}$ described in Section 3.1.2 that considers valid only executions of the form $(\texttt{on};\texttt{off})^{\star}$. Section 3.1.2 showed that, on systems in which all executions $(\texttt{on};\texttt{off})^{\star}$ are equivalent to the empty execution, $\hat{P}$ is effectively$_{\approx}$ but not effectively$_{=}$ enforceable by truncation automata. Likewise, $\hat{P}$ is effectively$_{\approx}$ but not effectively$_{=}$ enforceable by edit automata: an edit automaton can effectively$_{\approx}$ enforce $\hat{P}$ because a truncation automaton can effectively$_{\approx}$ enforce $\hat{P}$ (and edit automata can mimic truncation automata); an edit automaton cannot effectively$_{=}$ enforce $\hat{P}$ because doing so would require outputting verbatim all valid prefixes of the invalid input $(\texttt{on};\texttt{off})^{\omega}$, which would make the entire output equal to the invalid input.

Modifying the equivalence relation in this example to only consider finite executions of the form $(\texttt{on};\texttt{off})^{+}$ equivalent to $\texttt{on};\texttt{off}$ does not affect the fact that neither truncation nor edit automata can effectively$_{=}$ enforce $\hat{P}$ (because equivalence relations are irrelevant to effective$_{=}$ enforcement). However, the modified equivalence relation does prevent a truncation automaton T from effectively$_{\approx}$ enforcing $\hat{P}$ because T has no way to react to an initial on action (if T halts then it cannot output anything equivalent to the valid input $\texttt{on};\texttt{off}$, and if T accepts then it outputs an invalid sequence). An edit automaton E *can* effectively$_{\approx}$ enforce $\hat{P}$ with the modified equivalence relation, though, because E can respond to an initial $\texttt{on};\texttt{off}$ transaction by outputting $\texttt{on};\texttt{off}$ and then halting.

Comparing our analysis of the properties effectively$_{\approx}$ enforceable by edit automata with our analysis of the properties effectively$_{=}$ enforceable, we find that having to consider arbitrarily permissive equivalence relations (cf. Theorem 2.5) complicates the analysis greatly. Although it is interesting to note that even as effective$_{\approx}$ enforcers edit automata exercise their full enforcement powers by operating in a transactional manner, we believe that, as with truncation automata, the theorems related to effective$_{=}$ enforcement better capture the inherent power of edit automata. Effective$_{=}$ enforcement provides an elegant lower bound for what can be effectively$_{\approx}$ enforced in practice. In the future, it would be interesting to explore whether practically relevant constraints could be placed on semantic equivalence relations, and what effects such constraints would have on the powers of effective$_{\approx}$ enforcers.

4. INFINITE RENEWAL PROPERTIES

This section studies the class of 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 5.4 also describes two nonsafety renewal policies that we have enforced in an implemented system.

4.1 Renewal, Safety, and Liveness

Safety and Renewal. The most obvious way in which safety and 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 execution that satisfies a renewal property can be valid even if it has infinitely many invalid prefixes (as long as it also has infinitely many valid prefixes), but an execution satisfying a safety property can contain no invalid prefixes. Similarly, although infinite-length executions violating a renewal property can have prefixes that alternate a finite number of times between being valid and invalid, executions violating a safety property 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 $\neg\hat{P}(a)$.

Liveness and Renewal. There are renewal properties that are not liveness properties (e.g., the property that is only satisfied by the empty sequence, or, more generally, any safety property that considers at least one execution invalid), 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. As described in Section 2.2, every property is the conjunction of a single safety and a single liveness property. Following the strategy outlined in Section 2.2, the example renewal property satisfied only by $\cdot$ and $a;a$ decomposes into a safety property $\hat{P}_S$ satisfied exactly by executions $\cdot$, a, and $a;a$ and a liveness property $\hat{P}_L$ satisfied exactly by all finite executions except a. Although $\hat{P}_S$ is a renewal property, $\hat{P}_L$ is not (because invalid infinite-length executions have infinite valid prefixes).

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 more 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 3.3 in Appendix A shows how to construct an edit automaton (with a countably infinite state set) to enforce any of the renewal properties described in this subsection.

Renewal properties. Suppose we wish to constrain a user's interaction with a system that has actions a_1, a_2, and a_3, where a_3 ranges over all actions for opening files, a_2 over actions for logging out, and a_1 over all other actions. 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

$(a_1{}^\star; a_2)^\infty$.[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 discussed in Section 3.2.2, edit automata derive their power from being able to operate in a transactional manner. 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 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 $o_1; o_2; c_1; o_3; c_2; o_4; c_3; \ldots$,

[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.

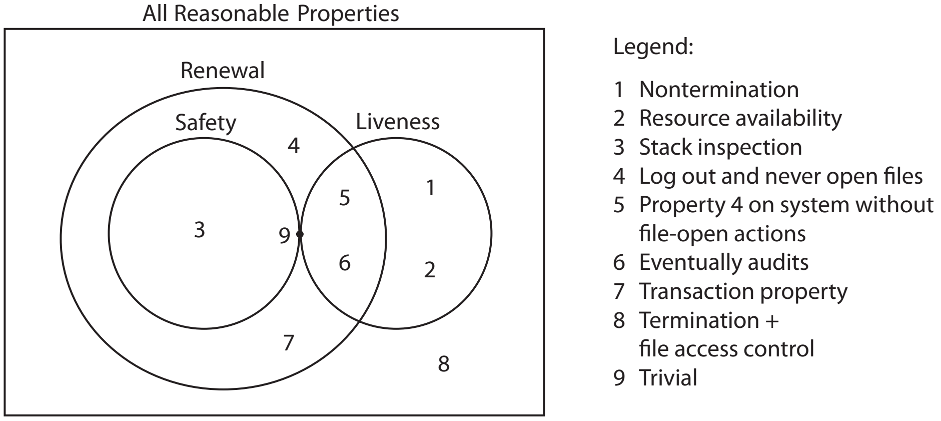

Fig. 1. Relationships between reasonable safety, liveness, and renewal properties.

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. LIMITATIONS AND PRACTICAL CONSIDERATIONS

Many gaps remain between real monitors and our models of them; we discuss the most obvious gaps in Sections 5.1–5.3. In Section 5.4, we describe Polymer, a language and system for enforcing run-time policies that was inspired by our work on edit automata and with which we have implemented monitors that enforce nonsafety policies. We discuss additional limitations of our model such as concurrency and distributed monitoring, which are the subject of ongoing research, in Section 6.2.

5.1 Computational and Resource Constraints

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.

Similarly, a system may contain actions uninsertable by monitors because, for example, the monitors (which may be firewalls, network scanners, or user-level operating-system extensions) lack access to secret keys that must be passed as parameters to the actions. In general, environmental factors beyond the control of the monitor may give rise to actions that are unsuppressible or uninsertable. In the future, we plan to explore the usefulness of explicitly defining, in the specification of systems, which actions are unsuppressible or uninsertable. This would allow us to describe more precisely the enforcement powers of monitors on those systems, though it would make the model significantly more complex. 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.

In addition, some actions of interest, although seemingly both insertable and suppressible, may have side effects that affect program behavior in undesired ways. For example, if an action a_1 updates a variable that is referenced by subsequent action a_2, then the effect of executing a_2 may be different depending on whether a_1 is suppressed or allowed to execute. Although software transactional memory [Shavit and Touitou 1995; Harris et al. 2005] could be used to ensure that the effects of suppressed memory operations are dealt with properly, in practice it may be intractable to reason about arbitrary compositions of suppressed effects, in which case many of these effectful actions might have to be considered unsuppressible or uninsertable.

Other practical constraints could be placed on monitors as well. For instance, Fong has shown that limiting the memory available to monitors induces limits on the properties they can enforce [Fong 2004]. Our models place no restrictions on the time and space available to monitors. In practice, however, it may be intractable to allow the monitors to use super-polynomial time or space to enforce a policy. Similarly, in many practical situations the time and space monitors can consume may be much more strictly limited (e.g., in real-time and embedded systems), and reasoning about which policies monitors can enforce in those circumstances would require models beyond what we have studied in this article.

5.2 Transactional Correctness

In addition to the difficulties suppressing or inserting individual actions described in Section 5.1, other problems may arise in practice that would prevent *sequences* of actions from being suppressed and inserted in a transactional manner. Much of the power of edit automata comes from their ability to suppress individually a sequence of dangerous actions and later, if the entire sequence is determined to be valid, atomically insert them; more precisely, the insertions must succeed, and they must succeed atomically. Truncation automata, on the other hand, do not need to output sequences of actions atomically (and indeed cannot, since they only accept individual actions).

Various tools, such as transactional file and database systems, might help practical implementations of edit-automata-based enforcement systems ensure that sequences of actions can be inserted atomically. In general, though, we cannot guarantee that a monitor modeled by an edit automaton will be able to insert a sequence of actions atomically, or even that a single insertion will succeed. An attempted insertion could be interrupted, for example, by a power failure or abrupt shutdown, or even by physical destruction of the machine executing the monitor. Some of these interruptions can be overcome through auxiliary technical means (e.g., by disabling interrupts or supplying backup power to the machine). Other interruptions (e.g., physical destruction of the computer the monitor is running on) may be beyond the monitor's ability to predict or react to. In the first case, we assume that appropriate technical safeguards are in place; otherwise, and in the second case, these interruptions are outside our model.

Generalizing from atomicity, to be sure that the monitors' suppress-insert transactions always complete correctly, they should also satisfy the other ACID properties [Elmasri and Navathe 1994]: consistency preservation (upon completion of the transaction the system must be in a consistent state), isolation (the effects of a transaction should not be visible to other concurrently executing transactions until the first transaction is committed), and durability or permanence (the effects of a committed transaction cannot be undone by a future failed transaction). As with atomicity, practical systems could violate these properties. For example, if the system manipulates the monitor's state (accidentally, maliciously, or due to malfunction) or fails to execute the monitor or its actions correctly, any of the ACID properties may be violated. In such a scenario, no monitor can make any guarantees about the policies it enforces.

In our model, which does not account for the violations discussed above and which is simplified by disregarding the possibility of concurrent target applications, the ACID properties are preserved. A monitor ensures consistency preservation by simply verifying that the sequences it atomically inserts are consistent, i.e., obey the property being enforced. A monitor ensures durability or permanence of a committed transaction by relying on the executing system to correctly execute the inserted actions. Once an action has been inserted by the automaton, the automaton can no longer touch it; furthermore, failed transactions (i.e., sequences that don't obey the property) cause nothing to be output and so no part of the transaction is externally observable. We only model the actions of a single agent and therefore isolation is trivially satisfied; we discuss extending our model to concurrent and distributed systems in Section 6.2.

5.3 Nonuniformity

Another reasonable, and more general, way to define security automata and their powers of enforcement would be to permit the case where input executions may form a *subset* of $\mathcal{A}^\infty$. Intuitively, this alternative model would capture the realistic possibility that a monitor may know a priori that it will never be given as input certain executions, e.g., because some other security mechanism such as a type checker has already ruled out those executions. This alternative model also has the potential to give security automata practically useful new powers, such as the ability to assume that every execution will include explicit `start` and `end` actions. In this case, we say that the set of possible input executions is *nonuniform*, in that it is a subset of the (uniform) set of all possible executions. Nonuniformity can be useful in practice to enforce properties that could not otherwise be enforced. For example, on a uniform system $S = \{\texttt{start}, \texttt{end}, \texttt{fopen f1}, \texttt{fclose f1}, \texttt{fopen f2}, \texttt{fclose f2}, \ldots\}$, an edit automaton cannot enforce a property requiring that all open files get closed before program termination (please see Section 4.2 for details). However, this property is easily enforceable on the nonuniform system in which all monitor-input executions begin with `start` and end with `end` actions. To enforce the desired policy on such a nonuniform system, the monitor simply maintains a list of all open files and inserts `fclose` actions to close all open files before allowing the `end` action to execute.

However, the effect of considering nonuniformity goes beyond what is practically useful. In the case of truncation automata, for example, nonuniformity would allow the automata to enforce more than just safety properties. Consider the nontermination property, which requires that every execution not terminate. Nontermination would not be enforceable by any security automaton on a uniform system. However, on a nonuniform system that guarantees that all input executions are nonterminating, a truncation automaton could "enforce" the nontermination property by accepting all actions. In this case some amount of the "enforcement" is done by the nonuniformity, so the monitor is no longer the sole, or even main, enforcement mechanism. By considering only uniform systems, we have tried in this article to explore the pure enforcement powers of monitors that receive no enforcement help from auxiliary constraints.

Considering nonuniformity also raises a host of other questions. For example, should only input sequences have restricted domains or should output sequences be similarly restricted? Considering nonuniformity would also nontrivially increase the complexity of the definitions, theorems, and proofs, as they would all have to distinguish between potentially different system, input-execution, and output-execution domains. For these reasons, we believe that nonuniformity is an interesting topic, but one that warrants a complete treatment that is beyond the scope of this article.

5.4 Polymer: An Implemented System for Enforcing Run-time Policies

In this subsection we give a brief overview of Polymer, a language and system for enforcing run-time policies. The design of the Polymer specification language was heavily influenced by the edit-automaton model discussed in this article; the language allows users to specify, in addition to safety properties, some nonsafety renewal properties. Here we will focus on Polymer as a tool for implementing edit automata; we refer interested readers to other papers for further Polymer details, including discussions of concurrency, semantics of the policy-specification language, performance and overhead measurements, and low-level implementation details [Bauer et al. 2005a; Ligatti 2006]. Polymer is fully implemented and available for download [Bauer et al. 2005b].

Polymer policies describe constraints on the behavior of Java programs. Polymer 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 immediately before and after any security-relevant action (i.e., method) executes. This technique of bytecode (or binary) rewriting to invoke run-time monitors is a common implementation strategy [Erlingsson 2004; Evans 2000; Hamlen 2006].

Users specify a Polymer policy by writing a new class that extends Polymer's base `Policy` class. The syntax for writing the new policy is almost identical to standard Java syntax, though Polymer does provide some extra syntactic constructs to make it easy to analyze security-relevant actions dynamically. When creating a new policy class, users may implement methods that specify whether and under what circumstances security-relevant actions may execute. A policy may respond to an about-to-be-executed target-application action (which we call a *trigger action*) by inserting other actions, suppressing the trigger action (by halting, raising an exception, or replacing the trigger action with a precomputed, "feigned" return value), or allowing the trigger action to execute undisturbed. Polymer also contains built-in support for composing policies.

Using the Polymer system, we have specified and enforced some nonsafety renewal policies on real programs. We next summarize those policies; additional details are in Ligatti's thesis [Ligatti 2006]. The first of our implemented nonsafety policies ensures that (hypothetical) ATM machines generate a proper log when dispensing cash. Consider a simple ATM system for dispensing cash that contains the following three methods.

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

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

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

Suppose we wish to require that the ATM machine's software properly logs all cash dispensations. We will consider an execution valid if and only if it has the form $(\texttt{logBegin}(n); \texttt{dispense}(n); \texttt{logEnd}(n))^\infty$. That is, valid executions are sequences of valid transactions, where each valid transaction consists of logging how much 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), and we enforce it in Polymer in the expected way, by suppressing preliminary `logBegin` and `dispense` actions until we are guaranteed that the current transaction is valid, at which point the suppressed actions get inserted.

The second of our implemented nonsafety policies ensures that Java applications writing to a file, possibly using multiple file-write operations, eventually give the file satisfactory contents (e.g., to ensure that the file obeys a required file format). An auxiliary predicate, passed as a parameter to the policy's constructor, determines whether a file's contents are satisfactory. The auxiliary predicate might not hold in the middle of a sequence of file writes but must be satisfied after a later write; hence, the file-contents policy is not a safety policy. Actually, if we assume the auxiliary predicate is satisfiable then the file-contents policy is a liveness policy because any invalid finite execution can be made valid by executing whatever file-write operations will satisfy the predicate. Similarly to the previous example, we enforce this nonsafety renewal property by suppressing (feigning) writes to files until we can ensure their validity, at which point we insert all suppressed writes.

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, this article makes three principal contributions. First, we have created a framework for reasoning about run-time policy enforcement that allows us to model monitors as transformers, rather than recognizers, of executions. 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, more powerful monitors can enforce 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 sometimes enforce nonsafety properties.

6.2 Future and Ongoing Work

We outline here several of the most interesting and potentially useful directions in which the work described in this article could be extended, some of which we are actively pursuing.

Formally Linking Edit Automata with Polymer Policies. Section 5.4 included an informal description of the ability of Polymer policies to implement edit automata: Polymer policies may respond to trigger actions by inserting other actions, suppressing the trigger action (by halting, raising an exception, or replacing the trigger action with a precomputed, "feigned" return value), or allowing the trigger action to execute undisturbed. This implementation is simple and intuitive, but it would be nice to formally prove a 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 a practical system like Polymer.

Compositions of Policies and Monitors. Security mechanisms typically operate in concert with other mechanisms; in fact, a mechanism's correctness often depends on the correctness of cooperating mechanisms. For example, a firewall may operate correctly only if the operating system's access-control mechanisms simultaneously correctly prevent tampering with the firewall's rule base or executable image.

We plan to explore theories and strategies for composing and decomposing general policies and enforcement mechanisms. To date the major research in this area has focused specifically on information-flow policies [Mantel 2002; McLean 1996], pure liveness policies [Alpern and Schneider 1987; Abadi and Lamport 1993], pure safety policies (which includes all access-control policies) [Alpern and Schneider 1987; Abadi and Lamport 1993; Bonatti et al. 2002; Schneider 2000], and systems in which all programs are implicitly assumed to terminate [Yu et al. 2007]. We would like to develop a more general understanding of policy and mechanism composition that would allow us to specify and analyze exactly how all sorts of mechanisms and policies compose, how and where conflicts arise, and metapolicies for resolving such conflicts. To allow for efficient and effective enforcement, we hope to be able to decompose a specified policy into a collection of subpolicies, some of which can be enforced statically, others that can be enforced by rewriting program code, and yet others that must be enforced at run time.

Modeling Concurrent and Distributed Run-time Monitors. In practice, enforcement mechanisms often operate concurrently to enforce a unified policy (e.g., distributed intrusion-detection mechanisms and policy enforcement in Grid computing). Most of the work done so far on modeling run-time monitors, including this article, focuses on enforcement systems in nonconcurrent settings, preventing us from modeling many common and useful systems and enforcement mechanisms.

For example, our current model ignores all networked or concurrent systems (i.e., target applications), as well as policies and mechanisms on such systems. Although this is a severe limitation, we have found much complexity so far just reasoning about enforcement in nonconcurrent environments. However, as we become more comfortable with our models, we would like to consider the possibility of concurrency. For now, we recognize that permitting application concurrency and distributed policies and monitors seems to create fundamental problems. For example, we may no longer be able to model system executions as simple sequences of actions (i.e., total orderings of actions) that induce state transitions. We may instead want to define executions as partial orderings of actions in which actions may execute concurrently. From this primitive definition, we would have to build up an entirely new model of policies and enforcement mechanisms as operators on partial orderings of actions.

6.3 Closing Remarks

Given their abundance and practicality as enforcement mechanisms, it seems strange that we have few sophisticated models for reasoning about monitors' actual enforcement capabilities. Even basic results, such as that practical monitors can sometimes enforce liveness properties, are surprisingly beyond the scope of previous models of strictly run-time enforcement mechanisms.

By continuing to explore the capabilities 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. In the long term, 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 efficient mechanisms that provably enforce given policies.

Acknowledgments. We are grateful to the anonymous referees for their constructive suggestions for improving earlier versions of this article. This research was supported by NSF grants CNS-0742736, CNS-0716343, and CNS-0716216, and by Army Research Office grant DAAD19-02-1-0389.

REFERENCES

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

Abadi, M. and Lamport, L. 1993. Composing specifications. *ACM Transactions on Programming Languages and Systems 15,* 1, 73–132.

Aktug, I., Dam, M., and Gurov, D. 2008. Provably correct runtime monitoring. In *Proceedings of the 15th International Symposium on Formal Methods.* 262–277.

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

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

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 2005 Conference on Programming Language Design and Implementation.*

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.

Bonatti, P., di Vimercati, S. D. C., and Samarati, P. 2002. An algebra for composing access control policies. *ACM Transactions on Information and System Security 5,* 1 (February), 1–35.

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.* 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 *Proceedings of the 5th ACM Conference on Computer and Communications Security.* 38–48.

Elmasri, R. and Navathe, S. B. 1994. *Fundamentals of database systems.* The Benjamin/Cummings Publishing Company, Inc.

Erlingsson, Ú. 2004. 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*. 87–95.

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

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 *Proceedings of the 1999 IEEE Symposium on Security and Privacy*.

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

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

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

Hamlen, K. W., Morrisett, G., and Schneider, F. B. 2006b. Certified in-lined reference monitoring on .NET. In *Proceedings of the 2006 workshop on Programming languages and analysis for security*. 7–16.

Harris, T., Marlow, S., Jones, S. L. P., and Herlihy, M. 2005. Composable memory transactions. In *Proceedings of the ACM Symposium on Principles & Practice of Parallel Programming*. 48–60.

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 *Proceedings of the Second International Workshop on Runtime Verification*.

Kim, M., Viswanathan, M., Ben-Abdallah, H., Kannan, S., Lee, I., and Sokolsky, O. 1999. Formally specified monitoring of temporal properties. In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*.

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 Transactions on Software Engineering 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 *Computer Security—ESORICS 2005: 10th European Symposium on Research in Computer Security*.

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.

Mantel, H. 2002. On the composition of secure systems. In *Proceedings of the IEEE Symposium on Security and Privacy*. 88–101.

Martinelli, F. and Matteucci, I. 2007a. An approach for the specification, verification and synthesis of secure systems. *Electronic Notes in Theoretical Computer Science 168*, 29–43.

Martinelli, F. and Matteucci, I. 2007b. Through modeling to synthesis of security automata. *Electronic Notes in Theoretical Computer Science 179*, 31–46.

Martinelli, F. and Mori, P. 2007. Enhancing Java security with history based access control. In *Foundations of Security Analysis and Design*.

Matteucci, I. 2006. A tool for the synthesis of programmable controllers. In *4th International Workshop on Formal Aspects in Security and Trust (FAST 2006)*.

Matteucci, I. 2007. Automated synthesis of enforcing mechanisms for security properties in a timed setting. *Electronic Notes in Theoretical Computer Science 186*, 101–120.

McLean, J. 1996. A general theory of composition for a class of possibilistic properties. *IEEE Transactions on Software Engineering 22,* 1, 53–67.

Milner, R. 1978. Synthesis of communicating behaviour. In *Mathematical Foundations of Computer Science*. Lecture Notes in Computer Science, vol. 64. 71–83.

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*. 3967–3976.

Schneider, F. B. 1987. Decomposing properties into safety and liveness using predicate logic. Tech. Rep. TR 87-874, Cornell University. Oct.

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 *Proceedings of the 26th International Conference on Software Engineering*. 418–427.

Shavit, N. and Touitou, D. 1995. Software transactional memory. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*. 204–213.

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*. 203–216.

Walker, D. 2000. A type system for expressive security policies. In *Twenty-Seventh ACM Symposium on Principles of Programming Languages*. 254–267.

Yu, D., Chander, A., Islam, N., , and Serikov, I. 2007. JavaScript instrumentation for browser security. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 237–249.

A. PROOFS

This appendix contains proofs for theorems presented in Section 3.

Theorem 3.1 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) $\forall\sigma \in \mathcal{A}^\infty : \neg\hat{P}(\sigma) \implies \exists\sigma'{\preceq}\sigma : \forall\tau{\succeq}\sigma' : \neg\hat{P}(\tau)$ (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 δ 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. □

Theorem 3.2 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 δ 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. □

THEOREM 3.3 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*

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:

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

This δ 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 remaining 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 $\hat{P}$. □

Theorem 3.4 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 there exists decidable* eager-insertion function $f:(\mathcal{A}^\star\times\mathbb{N})\rightarrow(\{\cdot\}\cup\mathcal{A})$ *such that the following constraints are met.*

(1) $\forall\sigma \in \mathcal{A}^\omega : \hat{P}(\sigma) \iff \begin{pmatrix} \forall\sigma'{\preceq}\sigma : \exists\tau{\preceq}\sigma : \sigma'{\preceq}\tau \wedge \hat{P}(\tau) \\ \vee\ \exists\sigma'{\preceq}\sigma : \sigma{=}\sigma'; f(\sigma',0); f(\sigma',1); f(\sigma',2); \ldots \end{pmatrix}$

(2) $\forall\sigma \in \mathcal{A}^\infty : \forall\sigma' \in \mathcal{A}^\star : (f(\sigma',0){\neq}\cdot \wedge \sigma{=}\sigma'; f(\sigma',0); f(\sigma',1); f(\sigma',2); \ldots) \implies$

(a) $\hat{P}(\sigma)$

(b) $\sigma \in \mathcal{A}^\star \implies (\forall\tau{\succeq}\sigma' : \hat{P}(\tau) \implies \sigma{\preceq}\tau)$

(c) $\sigma \in \mathcal{A}^\omega \implies (\forall\tau{\succeq}\sigma' : \hat{P}(\tau) \implies \sigma{=}\tau)$

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

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

PROOF. *(If Direction)* We construct 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 currently suppressed)

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

—Transition function (for simplicity, we write δ in terms of high-level transitions): Consider processing an action a in state (τ,σ).

(A). If $\neg\hat{P}(\tau;\sigma;a)$ and $f(\tau;\sigma;a,0) = \cdot$ then suppress a and continue in state $(\tau,\sigma;a)$.

(B). If $\hat{P}(\tau;\sigma;a)$ and $f(\tau;\sigma;a,0) = \cdot$ then insert $\sigma;a$ (one action at a time), suppress a, and continue in state $(\tau;\sigma;a,\cdot)$.

(C). If $f(\tau;\sigma;a,0) \neq \cdot$ then let $\sigma' = f(\tau;\sigma;a,0); f(\tau;\sigma;a,1); f(\tau;\sigma;a,2); \ldots$ and insert $\sigma;a;\sigma'$ (one action at a time; note that σ' may have infinite length, in which case the automaton enters an infinite loop and the rest of this transition description is irrelevant). Then suppress a. If, after suppressing a, the next actions input equal σ' then suppress each of those already inserted actions and continue in state $(\tau;\sigma;a;\sigma',\cdot)$; otherwise halt (i.e., suppress all additional input actions).

This δ is computable because f and $\hat{P}$ are decidable over finite-length executions.

E maintains the invariant $I_{\hat{P}}(q)$ on states $q = (\tau,\sigma)$ 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 $\hat{P}(\tau)$. The automaton can initially establish $I_{\hat{P}}(q_0)$ because $q_0 = (\cdot,\cdot)$, $(q_0,\cdot) \Downarrow_E \cdot$, 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. The inductive argument makes use of constraint (2b) in the theorem statement to show that E maintains the invariant after performing transition (C). Also, in transition (C), there are two possibilities for E to stop making new (high-level) transitions: (1) E may enter an infinite loop or (2) E may halt. For convenience, we call these Special Operations (1) and (2) below.

Next, we let $\sigma \in \mathcal{A}^\infty$ be an automaton input and show that E correctly effectively$_=$ enforces $\hat{P}$ on σ. First, for this and the following two paragraphs we only consider the case in which E never performs Special Operations (1) or (2) on input σ. If $\sigma \in \mathcal{A}^\star$ then by the automaton invariant, E on input σ will halt having output some τ such that $\hat{P}(\tau)$ and $\hat{P}(\sigma) \implies \sigma{=}\tau$.

If $\sigma \in \mathcal{A}^\omega$ and $\hat{P}(\sigma)$, constraint (1) in the theorem statement ensures that σ either has an infinite number of valid prefixes or has a prefix that extends to σ

using consecutive f-insertions. If σ has an infinite number of valid prefixes, the automaton invariant and transition function implies that E on σ correctly outputs every prefix of σ and only prefixes of σ. If, on the other hand, σ has a prefix σ' that extends to σ using consecutive f-insertions, E will perform Special Operation (1) in transition (C), something we are currently assuming does not occur.

If $\sigma \in \mathcal{A}^\omega$ and $\neg\hat{P}(\sigma)$, constraint (1) in the theorem statement ensures that σ has a finite number of valid prefixes and has no prefix that extends to σ using consecutive f-insertions (which implies that E does not perform Special Operation (1) on input σ, something we are already assuming). Because σ has a finite number of valid prefixes, we derive from the automaton invariant, the automaton transition function, and the definition of $\Downarrow_E$, that E on input σ outputs some τ' such that $\hat{P}(\tau')$. Note that τ' must be the longest valid prefix of σ because E cannot make transition (C) after outputting τ' unless it also performs Special Operation (1) or (2), which we are assuming does not occur. This is because constraint (2a) in the theorem statement and the automaton invariant together guarantee that the actions E inserts in transition (C), when concatenated with the actions E has already output, satisfy $\hat{P}$. Hence, if Special Operation (1) is not performed in transition (C) after E outputs τ' then Special Operation (2) must be performed because by assumption no extensions of τ' (besides τ' itself) are valid prefixes of σ.

Now consider the case in which E finishes processing σ with Special Operation (1). In this case constraints (2a) and (2c) in the theorem statement and the automaton invariant together guarantee that the actions E inserts in transition (C), when concatenated with the actions E has already output, satisfy $\hat{P}$ and are equal to any valid extension of the current input.

Finally consider the case in which E finishes processing σ with Special Operation (2). In this case constraints (2a) and (2b) in the theorem statement and the automaton invariant together guarantee that the actions E inserts in transition (C), when concatenated with the actions E has already output, satisfy $\hat{P}$ and are a prefix of any valid extension of the current input. However, the definition of transition (C) implies that because E performs Special Operation (2), E's input does not extend its output immediately before halting; therefore E's input must be invalid, yet its output satisfies $\hat{P}$. Hence, in this case and all the others above, E correctly effectively$_=$ enforces $\hat{P}$.

(Only-If Direction). For all $\sigma \in \mathcal{A}^\star$ and $n \in \mathbb{N}$, we define $f(\sigma, n)$ by running E on input σ. If E on input σ outputs any sequence of the form $\sigma; a_0; a_1; ..; a_n; \sigma'$ (for actions $a_0, .., a_n$ and action sequence $\sigma' \in \mathcal{A}^\star$) then $f(\sigma, n) = a_n$; otherwise $f(\sigma, n) = \cdot$. Eager-insertion function f is decidable because E has a decidable transition function and σ has finite length.

With the definition of f and the assumption that E effectively$_=$ enforces $\hat{P}$, we show that constraints (1)–(4) in the theorem statement hold. For constraint (1), first 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 constraint (1) is untrue for σ. This implies that there is some valid prefix τ of σ after which all longer prefixes of σ violate $\hat{P}$. After outputting τ 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 constraint (1) does not hold on σ by assumption, the construction of the f function above implies that no prefix of σ, when input to E, causes E to output σ in its entirety. Therefore, E cannot output any prefixes of σ after outputting τ, so E fails to effectively$_=$ enforce $\hat{P}$ on this σ. Our assumption was therefore incorrect, and constraint (1) must hold in this case.

Now consider any $\sigma \in \mathcal{A}^\omega$ such that $\neg\hat{P}(\sigma)$. Assume for the sake of obtaining a contradiction that constraint (1) does not hold on σ, implying that either (a) there are an infinite number of valid prefixes of σ or (b) there is a prefix σ' of σ such that $\sigma{=}\sigma'; f(\sigma',0); f(\sigma',1); f(\sigma',2); \ldots$. However, (b) cannot be true because if it were then the definition of f would imply that on invalid input σ, E outputs σ, something an effective enforcer cannot do. Also, (a) cannot be true because E is an effective$_=$ enforcer and can only enforce $\hat{P}$ on sequences obeying $\hat{P}$ by emitting them verbatim; if (a) were true then E on input σ would have to emit every one of the valid prefixes of σ, thereby outputting the invalid σ in full (again something an effective enforcer cannot do). Our assumption that constraint (1) does not hold was therefore incorrect, and in all cases constraint (1) must hold.

We next show that constraint (2) must also hold. Consider any $\sigma \in \mathcal{A}^\infty$ and $\sigma' \in \mathcal{A}^\star$ such that $f(\sigma',0) \neq \cdot$ and $\sigma{=}\sigma'; f(\sigma',0); f(\sigma',1); f(\sigma',2); \ldots$. By the definition of function f, E on input σ' outputs σ, $\sigma'{\preceq}\sigma$, $\sigma'{\neq}\sigma$, and $\neg\hat{P}(\sigma')$ (otherwise, effective$_=$ enforcer E could not output σ on input σ', given that $\sigma{\neq}\sigma'$). Because E is an effective enforcer, its output σ must be valid, so constraint (2a) holds. Also, if $\sigma \in \mathcal{A}^\star$ then the only way E could output σ on σ' yet still correctly effectively$_=$ enforce $\hat{P}$ on all valid extensions τ of σ' is for τ to begin with σ (otherwise E on input τ could not output τ verbatim). Hence, constraint (2b) holds. Similarly, if $\sigma \in \mathcal{A}^\omega$ then the only way E could output σ on σ' yet still correctly effectively$_=$ enforce $\hat{P}$ on all valid extensions τ of σ' is for there to be exactly one such τ, which must be equal to σ itself (otherwise E on valid input τ would output infinite-length σ rather than τ). Hence, constraint (2c) also holds.

Finally, constraints (3) and (4) hold for the same reasons given in the "Only-If" direction of the proof of Theorem 3.1. □

THEOREM 3.8 EFFECTIVE$_\approx$ E^∞-ENFORCEMENT. *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* transaction action-output function f *such that the following constraints are met.*

(1) $\forall(\sigma_0, .., \sigma_n) \in (\mathcal{A}^+)^\star : (n < 1 \ \vee\ O_f(\sigma_0, .., \sigma_{n-1}) \in \mathcal{A}^\star) \implies$
  - *(a)* $\hat{P}(O_f(\sigma_0, .., \sigma_n))$
  - *(b)* $\hat{P}(\sigma_0; ..; \sigma_n) \implies \sigma_0; ..; \sigma_n \approx O_f(\sigma_0, .., \sigma_n)$
  - *(c)* $\forall\tau \succ (\sigma_0; ..; \sigma_n) : (\hat{P}(\tau) \ \wedge\ O_f(\tau){=}O_f(\sigma_0, .., \sigma_n)) \implies \tau \approx O_f(\sigma_0, .., \sigma_n)$
  - *(d)* $\forall\tau \in \mathcal{A}^\infty : (\hat{P}(\tau) \ \wedge\ \sigma_0; ..; \sigma_n \prec \tau \ \wedge\ O_f(\sigma_0; ..; \sigma_n) \in \mathcal{A}^\omega) \implies$ $\tau \approx O_f(\sigma_0, .., \sigma_n)$

(2) $\forall(\sigma_0, \sigma_1, \ldots) \in (\mathcal{A}^+)^\omega : (\forall i \in \mathbb{N} : O_f(\sigma_0, .., \sigma_i) \in \mathcal{A}^\star) \implies$
  - *(a)* $\hat{P}(O_f(\sigma_0, \sigma_1, \ldots))$
  - *(b)* $\hat{P}(\sigma_0; \sigma_1; \ldots) \implies (\sigma_0; \sigma_1; \ldots) \approx O_f(\sigma_0, \sigma_1, \ldots)$

Proof. *(If Direction)* 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 sequence of valid transactions input so far paired with the sequence of actions currently suppressed)

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

—Transition function (for simplicity, we write δ in terms of high-level transitions): When processing an action a in state (S, σ), always make the following high-level transition. First, insert $\tau = F_f(S, \sigma; a)$ (one action at a time; note that τ may have infinite length, in which case the automaton enters an infinite loop and the rest of this transition description is irrelevant). Then suppress a and continue either in state $(S, (\sigma; a))$ if $\tau = \cdot$, or in state $((S, \sigma; a), \cdot)$ otherwise.
This δ is computable because part (2) of Definition 3.5 implies that $F_f(S, \sigma; a)$ is always computable (one output action at a time).

Let τ be $\bar{S}$, the concatenation of all executions in a sequence of executions S (where $S \in (\mathcal{A}^+)^\star$ and $\tau \in \mathcal{A}^\star$). Then E maintains the invariant $I_f(q)$ on states $q = (S, \sigma)$ that exactly $O_f(S) = O_f(S, \sigma)$ has been output, $\tau; \sigma$ is the input that has been processed, and $(q_0, \tau; \sigma) \Downarrow_E O_f(S)$. The automaton can initially establish $I_f(q_0)$ because $q_0 = (\cdot, \cdot)$, $O_f(\cdot) = \cdot$, and $(q_0, \cdot) \Downarrow_E \cdot$. A simple inductive argument on the transition relation suffices to show that E maintains the invariant in every state it reaches.

Next, we show that E correctly effectively$_\approx$ enforces $\hat{P}$ on any $\sigma \in \mathcal{A}^\infty$. First consider the case in which E on input σ never enters an infinite loop, as described in the high-level transition function above. Let the prefixes of σ for which F_f does not return $\cdot$ be laid out as a finite or infinite sequence of nonempty executions $(\sigma_0, \sigma_1, \ldots)$ such that $(\sigma_0; \sigma_1; \ldots) \preceq \sigma$ and $F_f(\sigma_0) \neq \cdot$, $F_f(\sigma_0; \sigma_1) \neq \cdot$, etc. By E's transition function and invariant, $(q_0, \sigma) \Downarrow_E O_f(\sigma_0, \sigma_1, \ldots)$. Therefore, if $(\sigma_0; \sigma_1; \ldots) = \sigma$ then, regardless of whether $(\sigma_0; \sigma_1; \ldots)$ is a finite or infinite sequence, constraints (1a), (1b), (2a), and (2b) ensure that E correctly effectively$_\approx$ enforces $\hat{P}$ on σ. Also, if $(\sigma_0; \sigma_1; \ldots) \neq \sigma$ (which implies that $(\sigma_0; \sigma_1; \ldots)$ is a finite sequence of executions), constraint (1a) implies that E's output $O_f(\sigma_0, \sigma_1, \ldots)$ satisfies $\hat{P}$, while constraint (1c) implies that if $\hat{P}(\sigma)$ then $\sigma \approx O_f(\sigma_0, \sigma_1, \ldots)$. Again, E correctly effectively$_\approx$ enforces this σ.

On the other hand, if E on input σ does enter an infinite loop after having input $\tau \preceq \sigma$, the automaton invariant implies that E, while considering some input action a, ceases to input further actions of σ in some state (S, σ') such that $\tau = \bar{S}; \sigma'; a$. By the automaton invariant and transition function, E's (infinite-length) output must be $O_f(S, \sigma; a)$, so $(q_0, \sigma) \Downarrow_E O_f(S, \sigma; a)$. By constraint (1a) in the theorem statement, $\hat{P}(O_f(S, \sigma; a))$. Also, by constraint (1d), $\hat{P}(\sigma) \implies \sigma \approx O_f(S, \sigma; a)$. Hence, in all cases E correctly effectively$_\approx$ enforces $\hat{P}$.

(Only-If Direction). For all $(\sigma_0, .., \sigma_n) \in (\mathcal{A}^+)^\star$ and $m \in \mathbb{N}$, define $f((\sigma_0, .., \sigma_n), m)$ recursively. As a basis, $f(\cdot, m) = \cdot$. In the general case, first use the recursive definitions of $f((\sigma_0, .., \sigma_{n-1}), 0)$, $f((\sigma_0, .., \sigma_{n-1}), 1)$, $\ldots$ to calculate $\tau = O_f(\sigma_0, .., \sigma_{n-1})$. If τ has infinite length then the value returned by f is irrelevant. Assuming τ is finite, run E on input $\sigma = \sigma_0; ..; \sigma_n$ until τ has been output (τ must be output because our definition of f maintains the invariant that E on any input

$\sigma_0;..;\sigma_j \in (\mathcal{A}^+)^\star$ outputs $O_f(\sigma_0,..,\sigma_j)$). After E outputs τ on input σ, define $f((\sigma_0,..,\sigma_n),0)$ to be the next action output after τ (or $\cdot$ if nothing more than τ is output), $f((\sigma_0,..,\sigma_n),1)$ to be the second action output after τ (or $\cdot$ if less than two actions are output after τ), etc. This f satisfies all four constraints in Definition 3.5 and is therefore a valid transaction action-output function.

With the definition of f, the invariant that E on any input $\sigma_0;..;\sigma_j \in (\mathcal{A}^+)^\star$ outputs $O_f(\sigma_0,..,\sigma_j)$, and the assumption that E effectively$_\approx$ enforces $\hat{P}$, we show that constraints (1) and (2) in the theorem statement hold. For constraint (1) we consider a sequence of executions $S = (\sigma_0,..,\sigma_n) \in (\mathcal{A}^+)^\star$ such that E outputs only finitely many actions in response to $\sigma_0;..;\sigma_{n-1}$ (or in response to $\cdot$ when S is a sequence of zero or one executions). Let $\sigma = \sigma_0;..;\sigma_n$. The invariant on E and the fact that E effectively$_\approx$ enforces $\hat{P}$ imply constraints (1a) and (1b). Moreover, if E fails to output anything beyond $O_f(S)$ when input a strict extension of σ then the definition of effective$_\approx$ enforcement implies constraint (1c). Finally, if E outputs an infinite-length execution on input σ (i.e., $O_f(S) \in \mathcal{A}^\omega$) then the definition of effective$_\approx$ enforcement implies constraint (1d).

For constraint (2) we consider a sequence of executions $S = (\sigma_0, \sigma_1, \ldots) \in (\mathcal{A}^+)^\omega$ such that E outputs only finitely many actions in response to $\sigma_0;..;\sigma_i$, for all $i \in \mathbb{N}$. Let $\sigma = \sigma_0;\sigma_1;\ldots$. We can extend the invariant on E (stipulating that E on any input $\sigma_0;..;\sigma_j \in (\mathcal{A}^+)^\star$ outputs $O_f(\sigma_0,..,\sigma_j)$) to the infinite sequence S in this case by the definition of O_f (Definition 3.7). Because E on input σ outputs $O_f(S)$, and E effectively$_\approx$ enforces $\hat{P}$, constraints (2a) and (2b) must also hold. □