International Journal of Information Security manuscript No.
(will be inserted by the editor)

Edit Automata: Enforcement Mechanisms for Run-time Security Policies*

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

[1] Princeton University, Princeton, NJ, USA, e-mail: {jligatti,dpw}@cs.princeton.edu
[2] Carnegie Mellon University, Pittsburgh, PA, USA, e-mail: lbauer@cmu.edu

Abstract We analyze the space of security policies that can be enforced by monitoring and modifying programs at run time. Our program monitors, called *edit automata*, are abstract machines that examine the sequence of application program actions and transform the sequence when it deviates from a specified policy. Edit automata have a rich set of transformational powers: They may terminate the application, thereby truncating the program action stream; they may suppress undesired or dangerous actions without necessarily terminating the program; and they may also insert additional actions into the event stream.

After providing a formal definition of edit automata, we develop a rigorous framework for reasoning about them and their cousins: *truncation automata* (which can only terminate applications), *suppression automata* (which can terminate applications and suppress individual actions), and *insertion automata* (which can terminate and insert). We give a set-theoretic characterization of the policies each sort of automaton can enforce, and we provide examples of policies that can be enforced by one sort of automaton but not another.

Key words Run-time checking and monitoring – security automata – classification of security policies – language-based security

1 Introduction

When designing a secure, extensible system such as an operating system that allows applications to download code into the kernel or a database that allows users to submit their own optimized queries, we must ask two important questions.

1. What sorts of security policies can we expect our system to enforce?
2. What sorts of mechanisms do we need to enforce these policies?

Neither of these questions can be answered effectively without understanding the space of enforceable security policies and the power of various enforcement mechanisms.

The first significant effort to define the class of enforceable security policies is due to Schneider [Sch00]. He investigated the security properties that can be enforced by a specific type of program monitor. One of Schneider's monitors can interpose itself between an untrusted program and the machine on which the program runs. The monitor can examine the sequence of security-relevant program actions one at a time, and if it recognizes an action that will violate its policy, the monitor terminates the program. This mechanism is very general since decisions about whether or not to terminate the program can depend upon the entire history of the program's execution. However, since these monitors can only terminate programs and cannot otherwise modify their behavior, it is possible to define still more powerful enforcement mechanisms.

In this paper, we re-examine the question of which security policies can be enforced at run time by monitoring untrusted programs. Our overall approach differs from Schneider's, who also used automata to model program monitors, in that we view program monitors as transformers that *edit* the stream of actions produced by an untrusted application. This new viewpoint leads us to define a hierarchy of enforcement mechanisms, each with different transformational capabilities:

– A *truncation automaton* can recognize bad sequences of actions and halt program execution before the security policy is violated, but cannot otherwise modify program behavior. These automata are similar to Schneider's original security monitors.

* This is a revised and extended version of "More Enforceable Security Policies," a paper that first appeared in the Workshop on Foundations of Computer Security, June 2002 [BLW02].

- A *suppression automaton*, in addition to being able to halt program execution, has the ability to suppress individual program actions without terminating the program outright.
- An *insertion automaton* is able to insert a sequence of actions into the program action stream as well as terminate the program.
- An *edit automaton* combines the powers of suppression and insertion automata. It is able to truncate action sequences and insert or suppress security-relevant actions at will.

We use the general term *security automaton* to refer to any automaton that is used to model a program monitor, including the automata mentioned above.[1]

The main contribution of this article is the development of a robust theory for reasoning about these machines under a variety of different conditions. We use our theory to characterize the class of security policies that can be enforced by each sort of automaton, and we provide examples of security policies that lie in one class but not another.

More important than any particular result is our methodology, which gives rise to straightforward, rigorous proofs concerning the power of security mechanisms and the range of enforceable security policies. This overall methodology can be broken down into four main parts.

Step 1. Define the underlying computational framework and the range of security policies that will be considered. In this paper, we define the software systems and sorts of policies under consideration in Section 2.1 through Section 2.3.

Step 2. Specify what it means to enforce a security policy. As we will see in Section 2.4, there are several choices to be made in this definition. One must be sure that the enforcement model accurately reflects the desires of the system implementer and the environment in which the monitor operates. Section 2.5 explains some of the limitations induced by our decisions in steps 1 and 2.

Step 3. Formally specify the operational behavior of the enforcement mechanism in question. Sections 3, 4, 5, and 6 define the operational semantics of four different sorts of monitors and provides examples of the policies that they can enforce.

Step 4. Prove from the previous definitions that the security mechanism in question is able to enforce the desired properties. Sections 3, 4, 5, and 6 state theorems concerning the security policies that each type of monitor can enforce. The formal proofs can be found in Appendix A.

After completing our analysis of edit automata and related machines, we discuss related work (Section 7). Finally, Section 8 concludes the paper with a taxonomy of security policies and a discussion of some unanswered questions and our continuing research.

2 Security Policies and Enforcement Mechanisms

In this section, we define the overarching structure of the secure systems we intend to explore. We also define what it means to be a security policy, and what it means to enforce a security policy. Finally, we give a generic definition of a security automaton as an action sequence transformer.

2.1 Systems, Executions, and Policies

We specify software systems at a high level of abstraction; a system is specified via a set of *program actions* $\mathcal{A}$ (also referred to as program events). An *execution* σ is simply a finite sequence of actions $a_1, a_2, \ldots, a_n$. Previous authors [Sch00] have considered infinite executions as well as finite ones because some interesting applications on which we might want to enforce policies (such as web servers or operating systems) are often considered to run infinitely. Although we allow a system's action set $\mathcal{A}$ to be countably infinite, in this paper we restrict ourselves to finite but arbitrarily long executions in order to simplify our analysis. Because we consider every step of a program to be a program action, limiting the analysis to finite executions implies that we are focusing on programs that terminate, leaving the analysis of nonterminating programs for future work. We use the notation $\mathcal{A}^\star$ to denote the set of all finite-length sequences of actions on a system with action set $\mathcal{A}$. In addition, we use the metavariables σ and τ to range over executions (i.e., elements of $\mathcal{A}^\star$) and the metavariable Σ to range over sets of executions (i.e., subsets of $\mathcal{A}^\star$).

The symbol $\cdot$ denotes the empty sequence. We use the notation $\sigma[i]$ to denote the i^{th} action in the sequence (beginning the count at 0). The notation $\sigma[..i]$ denotes the subsequence of σ involving the actions $\sigma[0]$ through $\sigma[i]$, and $\sigma[i+1..]$ denotes the subsequence of σ involving all other actions. We use the notation $\tau;\sigma$ to denote the concatenation of two sequences. When τ is a prefix of σ we write $\tau \preceq \sigma$.

A *security policy* is a predicate P on sets of executions. A set of executions Σ satisfies a policy P if and only if $P(\Sigma)$. Most common extensional program properties fall under this definition of security policy, including the following.

[1] Previous authors [Sch00] have used the term to refer specifically to automata with powers similar to our truncation automata, which we discuss in Section 3.

- *Access Control* policies specify that no execution may operate on certain resources such as files or sockets, or invoke certain system operations.
- *Availability* policies specify that if a program acquires a resource during an execution, then it must release that resource at some (arbitrary) later point in the execution.
- *Bounded Availability* policies specify that if a program acquires a resource during an execution, then it must release that resource by some fixed point later in the execution. For example, the resource must be released in at most ten steps or after some system invariant holds. We call the condition that demands release of the resource the *bound* for the policy.
- An *Information Flow* policy concerning inputs s_1 and outputs s_2 might specify that if $s_2 = f(s_1)$ in one execution (for some function f) then there must exist another execution in which $s_2 \neq f(s_1)$.

2.2 Security Properties

Alpern and Schneider [AS87] distinguish between *properties* and more general policies as follows. A security policy P is deemed to be a (computable) *property* when it is a predicate over sets $\Sigma \subseteq \mathcal{A}^\star$ with the following form.

$$P(\Sigma) = \forall \sigma \in \Sigma . \hat{P}(\sigma) \qquad \text{(PROPERTY)}$$

where $\hat{P}$ is a computable predicate on $\mathcal{A}^\star$.

Hence, a property is defined exclusively in terms of individual executions and may not specify a relationship between different executions of the program. Information flow, for example, which can only be specified as a condition on the set of possible executions of a program, is not a property. The other example policies provided in the previous section are all security properties.

We assume that the empty sequence is contained in any property. This describes the idea that an untrusted program that has not started executing is not yet in violation of any property. From a technical perspective, this decision allows us to avoid repeatedly considering the empty sequence as a special case of an execution sequence in future definitions of enforceable properties.

Given some set of actions $\mathcal{A}$, a predicate $\hat{P}$ over $\mathcal{A}^\star$ induces the security property $P(\Sigma) = \forall \sigma \in \Sigma . \hat{P}(\sigma)$. We often use the symbol $\hat{P}$ interchangeably as a predicate over execution sequences and as the induced property. Normally, the context will make clear which meaning we intend.

Safety Properties Properties that specify that "nothing bad ever happens" are called *safety properties* [Lam77]. We can make this definition precise as follows. Predicate $\hat{P}$ induces a safety property on a system with action set $\mathcal{A}$ if and only if,

$$\forall \sigma \in \mathcal{A}^\star . \neg \hat{P}(\sigma) \Rightarrow \forall \tau \in \mathcal{A}^\star . \neg \hat{P}(\sigma ; \tau) \qquad \text{(SAFETY)}$$

Informally, this definition states that once a bad action has taken place, thereby excluding the initial segment of an execution from the property, there is no extension of that segment that can remedy the situation. For example, access-control policies are safety properties since once a restricted resource has been accessed, the policy is broken. There is no way to "un-access" the resource and fix the situation afterwards.

2.3 Security Automata

One way of enforcing security properties is with a *monitor* that runs in parallel with a *target* program [Sch00]. Whenever the target program wishes to execute a security-relevant operation, the monitor first checks its policy to determine whether or not that operation is allowed. If the target program's execution sequence σ is not in the property, the monitor transforms it into a sequence σ' that obeys the property.

A program monitor can be formally modeled by a *security automaton* A, which is a deterministic finite or countably infinite state machine $(Q, q_0, \mathcal{T})$ that is defined with respect to some system with action set $\mathcal{A}$. Q specifies the possible automaton states, and q_0 is the initial state. Each variety of automata will have a slightly different sort of transition function $\mathcal{T}$, and these differences account for the variations in their expressive power. The exact specification of $\mathcal{T}$ is part of the definition of each kind of automaton; we only require that $\mathcal{T}$ be complete, deterministic, and Turing Machine computable.

We will specify the execution of each different kind of security automaton A using a labeled operational semantics. The basic single-step judgment will have the form $(\sigma, q) \xrightarrow{\tau}_A (\sigma', q')$ where σ denotes the sequence of actions that the target program wants to execute; q denotes the current state of the automaton; σ' and q' denote the action sequence and state after the automaton takes a single step; and τ denotes the sequence of actions produced by the automaton. The input sequence σ is not observable to the outside world whereas the outputs, τ, are observable.

We generalize the single-step judgment to a multi-step judgment as follows. The basic multi-step judgment has the form $(\sigma, q) \stackrel{\tau}{\Longrightarrow}_A (\sigma', q')$.

$$\boxed{(\sigma, q) \stackrel{\tau}{\Longrightarrow}_A (\sigma', q')}$$

$$\frac{}{(\sigma, q) \stackrel{\cdot}{\Longrightarrow}_A (\sigma, q)} \qquad \text{(A-REFLEX)}$$

$$\frac{(\sigma, q) \xrightarrow{\tau_1}_A (\sigma'', q'') \quad (\sigma'', q'') \stackrel{\tau_2}{\Longrightarrow}_A (\sigma', q')}{(\sigma, q) \stackrel{\tau_1;\tau_2}{\Longrightarrow}_A (\sigma', q')} \qquad \text{(A-TRANS)}$$

In this paper we consider security automata whose transition functions take the current state and input action (the first of the actions that the target program wants to execute) and returns a new automaton state and the sequence of actions the automaton outputs. Our automata may or may not consume the current input action while making a transition.

2.4 Enforceable Properties

To be able to compare and contrast the power of various security mechanisms, it is crucial to have a rigorous definition of what it means for a mechanism to enforce a security property. We believe that enforcement mechanisms can only accomplish their task effectively when they obey the following two abstract principles.

(SOUNDNESS) An enforcement mechanism must ensure that all observable outputs obey the property in question.

(TRANSPARENCY) An enforcement mechanism must preserve the semantics of executions that already obey the property in question.

The first criterion requires that our automata transform bad program executions in such a way that their observable outputs obey the property. A security mechanism that allows outsiders to observe executions that do not satisfy the given policy is an unsound and inadequate mechanism. The second criterion requires that enforcement mechanisms operate invisibly on executions that obey the property in question. In other words, any editing operations that they perform must not change the meaning of a valid execution.

Conservative Enforcement. When an enforcement mechanism is able to obey the first criterion but not necessarily the second, we say that the mechanism enforces the property *conservatively.*

Definition 1 (Conservative Enforcement) *An automaton A with starting state q_0* conservatively enforces *a property $\hat{P}$ on the system with action set $\mathcal{A}$ if and only if $\forall \sigma \in \mathcal{A}^\star\ \exists q'\ \exists \sigma' \in \mathcal{A}^\star$.*

1. $(\sigma, q_0) \stackrel{\sigma'}{\Longrightarrow}_A (\cdot, q')$, *and*
2. $\hat{P}(\sigma')$

Conservative enforcement gives the automaton great latitude. The relationship between the automaton's output and the output of the target is not constrained, so the automaton may choose to enforce policies simply by always outputting a particular stream of actions (or nothing at all), regardless of the behavior of the target. Such an automaton is able to enforce any property, but often not in a useful manner. The existential quantification of q' and σ' ensures that the automaton eventually halts on all inputs.

Precise Enforcement. In order to formalize the second enforcement criterion (TRANSPARENCY), we must be specific about what it means to preserve the semantics of a program execution. However, since this abstract notion may differ substantially from one computational context to the next, we offer a couple of different definitions. The simplest reasonable definition requires that the automaton in question outputs program actions in lockstep with the target program's action stream.

Definition 2 (Precise Enforcement) *An automaton A with starting state q_0* precisely enforces *a property $\hat{P}$ on the system with action set $\mathcal{A}$ if and only if $\forall \sigma \in \mathcal{A}^\star\ \exists q'\ \exists \sigma' \in \mathcal{A}^\star$.*

1. $(\sigma, q_0) \stackrel{\sigma'}{\Longrightarrow}_A (\cdot, q')$,
2. $\hat{P}(\sigma')$, *and*
3. $\hat{P}(\sigma) \Rightarrow \forall i\ \exists q''.\ (\sigma, q_0) \stackrel{\sigma[..i]}{\Longrightarrow}_A (\sigma[i+1..], q'')$

An automaton precisely enforces a property if and only if it conservatively enforces the property and outputs actions in lockstep with the target application. The automaton must not interfere with program execution in any way when the input obeys the property; every action in a valid input sequence must be accepted before the next action is considered. Precise enforcement is often the right correctness criterion for security monitors when any delay in outputting program actions will change the semantics of the application. This is usually the case in interactive applications that require an action to return a value before continuing.

Effective Enforcement. Our definition of precise enforcement does not recognize the fact that it is often the case that two syntactically different action sequences may be semantically equivalent. This often happens in three sorts of ways:

1. A sequence containing unnecessary actions or a series of idempotent actions may be equivalent to a sequence in which one or more of these actions are removed. In some systems, for example, closing a file twice in succession is equivalent to closing it just once.
2. A sequence which replaces some action with a different action that has the same semantics may be considered equivalent to the original sequence. It may also be permissible to replace several actions with a single action that subsumes their behavior. For example, an action that opens a socket followed by an action that sends data on the socket is semantically equivalent to a macro instruction that does both.
3. A sequence in which two or more independent actions are permuted may be equivalent to the original sequence. For example, if we want to open two different files, the order in which we do this may not matter.

Our final definition of enforcement uses a system-specific equivalence relation ($\cong$) on executions to take these possibilities into account. We require that the relation be reflexive, symmetric, and transitive. Moreover, any property that we might consider should not distinguish equivalent sequences:

$$\sigma \cong \sigma' \Rightarrow \hat{P}(\sigma) \Leftrightarrow \hat{P}(\sigma') \qquad \text{(EQUIVALENCE)}$$

With our equivalence relation in hand, we can give a concrete definition of what it means for a security automaton to *effectively enforce* a security property.

Definition 3 (Effective Enforcement) *An automaton A with starting state q_0* effectively enforces *a property $\hat{P}$ on the system with action set $\mathcal{A}$ if and only if* $\forall \sigma \in \mathcal{A}^\star\ \exists q'\ \exists \sigma' \in \mathcal{A}^\star$.

1. $(\sigma, q_0) \stackrel{\sigma'}{\Longrightarrow}_A (\cdot, q')$,
2. $\hat{P}(\sigma')$, *and*
3. $\hat{P}(\sigma) \Rightarrow \sigma \cong \sigma'$

Informally, an automaton effectively enforces a property whenever it conservatively enforces the property and obeys the principle of transparency.

2.5 Limitations

As always when reasoning about security, one must be acutely aware of situations in which formal results do not apply and of attacks that can come from outside the model. In our case, there are several realistic situations in which security automata fail to enforce a desired policy.

- The policy is not a predicate on execution sequences. If the policy depends upon some external environmental factors, such as the value of some secret that has been hidden from the monitor, the monitor may not be able to enforce the policy properly. Moreover, because monitors only see individual sequences of actions and cannot in general base decisions on other executions of the target application, security automata are enforcers of *properties* rather than general *policies*.

- The monitor is unable to effectively manipulate (insert, suppress, etc.) certain security-relevant actions. In a real-time system, some properties might not be enforceable because the monitor simply cannot perform the expected computation in the necessary real-time window. Alternatively, there may be some data (secret keys, passwords, etc.) that the monitor cannot effectively synthesize. This might mean that the monitor is unable to insert certain program actions and enforce certain properties.

- The monitor is unable to observe certain security-relevant actions. If the application has direct access to some hardware device and the monitor is unable to interpose itself between the application and the device, it may be unable to enforce certain properties.

- The monitor is compromised by the untrusted program. If the application is able to corrupt the monitoring code or data then the monitor will not be able to enforce any meaningful properties. In practice, this means that any software monitor must be isolated from the application using safe language technology or operating system support.

3 Truncation Automata

The first and most limited sort of security automaton that we will investigate is the *truncation automaton*. A truncation automaton does not change the target's output in any way unless the target attempts to invoke a forbidden operation. If this occurs, the truncation automaton halts the target program. This sort of automaton is the primary focus of Schneider's work [Sch00]. However, he viewed truncation automata as sequence recognizers. To fit this idea into our framework, we recast these machines as sequence transformers.

3.1 Definition

A truncation automaton T is a finite or countably infinite state machine (Q, q_0, δ) that is defined with respect to some system with action set $\mathcal{A}$. Q specifies the possible automaton states, and q_0 is the initial state. The partial function $\delta : \mathcal{A} \times Q \to Q$ specifies the transition function for the automaton and indicates that the automaton should accept the current input action and move to a new state. On inputs for which δ is not defined, the automaton halts the target program.

The operational semantics of truncation automata is specified below.

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

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

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

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

otherwise

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

3.2 Precisely Enforceable Properties

Erlingsson and Schneider [UES99, UES00] demonstrate that mechanisms similar to truncation automata can precisely enforce important access-control properties including software fault isolation and Java stack inspection. Interestingly, although the definition of truncation automata is independent of the definition of safety, truncation automata can precisely enforce exactly the set of safety properties.

Theorem 1 (Precise T-Enforcement) *A property $\hat{P}$ on a system with action set $\mathcal{A}$ can be precisely enforced by some truncation automaton if and only if $\forall \sigma \in \mathcal{A}^\star$. $\neg\hat{P}(\sigma) \Rightarrow \forall \tau \in \mathcal{A}^\star.\neg\hat{P}(\sigma;\tau)$.*

Proof Please see Appendix A for the proof. We omit it here due to its length. □

3.3 Effectively Enforceable Properties

Truncation automata can take advantage of a system's equivalence relation to effectively enforce non-safety properties.

Consider a system whose only actions are to open and close a login window. We wish to enforce the policy that the first action in any sequence must close the login window. The policy also allows the login window to be closed and then opened and closed any number of times, as this is considered equivalent to closing the window once. All sequences (and only those sequences) of the form $\mathsf{close};(\mathsf{open};\mathsf{close})^\star$ satisfy the property, and $\mathsf{close};(\mathsf{open};\mathsf{close})^\star \cong \mathsf{close}$. This is not a safety property because there is an illegal sequence ($\mathsf{close};\mathsf{open}$) that can be extended into a legal sequence ($\mathsf{close};\mathsf{open};\mathsf{close}$).

A truncation automaton can effectively enforce this property by checking that the first action closes the login window. If it does then the automaton accepts the action and halts; otherwise, it does not accept but still halts. This effectively enforces the property because any legal sequence will be equivalent to the accepted close action and all sequences emitted by the automaton (either $\cdot$ or close) obey the property.

This example illustrates both that truncation automata can effectively enforce more than just safety properties and that these non-safety properties are very limited. Before we can formally characterize the properties effectively enforceable by truncation automata, we must generalize the transformations truncation automata induce on instruction sequences into functions that act over sequences of actions. Given a set of actions $\mathcal{A}$, a function $\alpha^\star : \mathcal{A}^\star \to \mathcal{A}^\star$ is a *truncation-rewrite function* if it satisfies the following conditions.

1. $\alpha^\star(\cdot) = \cdot$
2. $\forall \sigma \in \mathcal{A}^\star\ \forall a \in \mathcal{A}.$
   $\alpha^\star(\sigma;a) = \alpha^\star(\sigma);a$, or
   $\alpha^\star(\sigma;a) = \alpha^\star(\sigma) \wedge H(\alpha^\star,\sigma;a)$

where $H(\rho^\star,\sigma)$ is a predicate defined on a general rewrite function $\rho^\star$ and a sequence σ as follows.

$$H(\rho^\star,\sigma) \Leftrightarrow (\forall \sigma' \in \mathcal{A}^\star.\ \sigma \preceq \sigma' \Rightarrow \rho^\star(\sigma') = \rho^\star(\sigma))$$

Informally, $H(\rho^\star,\sigma)$ is true if and only if an automaton whose output matches that of the rewrite function $\rho^\star$ halts on input σ, ceasing to output any additional actions. That is, whenever an automaton stops processing its input σ (for example, by applying rule T-Stop), it cannot examine any extensions to σ, so its output on any such extension must match its output on σ alone.

The restrictions placed on truncation-rewrite functions capture the operational restrictions of truncation automata. Condition (1) above ensures that truncation-rewrite functions do not output actions without receiving any input, and condition (2) stipulates that when examining the current action a, the rewrite function must either accept and output a (after it has finished outputting actions corresponding to earlier input) or halt and output nothing more. Note that when deciding how to transform an action a, the rewrite function can base its decision on the history of the entire execution up to a. Hence the input to the function in condition (2) is $\sigma;a$ and not just a.

The following theorem specifies the properties effectively enforceable by truncation automata.

Theorem 2 (Effective T-Enforcement) *A property $\hat{P}$ on a system with action set $\mathcal{A}$ can be effectively enforced by some truncation automaton if and only if there exists a computable truncation-rewrite function $\alpha^\star$ such that for all executions $\sigma \in \mathcal{A}^\star$,*

1. *If $\neg\hat{P}(\sigma)$ then $\hat{P}(\alpha^\star(\sigma))$*
2. *If $\hat{P}(\sigma)$ then $\sigma \cong \alpha^\star(\sigma)$*

Proof See Appendix A. □

4 Suppression Automata

Given our novel view of security automata as sequence transformers, it is a short step to define new sorts of automata that have greater transformational capabilities than truncation automata. In this section, we describe *suppression automata* and characterize the properties they enforce precisely and effectively.

4.1 Definition

A *suppression automaton* S is a finite or countably infinite state machine (Q, q_0, δ, ω) that is defined with respect to some system with action set $\mathcal{A}$. As before, Q is the set of all possible machine states, q_0 is a distinguished starting state for the machine, and the partial function δ specifies the transition function. The partial function $\omega : \mathcal{A} \times Q \to \{-,+\}$ has the same domain as δ

and indicates whether or not the action in question is to be suppressed ($-$) or emitted ($+$).

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

$$(\sigma, q) \xrightarrow{a}_S (\sigma', q') \qquad \text{(S-StepA)}$$
if $\sigma = a; \sigma'$
and $\delta(a, q) = q'$
and $\omega(a, q) = +$

$$(\sigma, q) \xrightarrow{\cdot}_S (\sigma', q') \qquad \text{(S-StepS)}$$
if $\sigma = a; \sigma'$
and $\delta(a, q) = q'$
and $\omega(a, q) = -$

$$(\sigma, q) \xrightarrow{\cdot}_S (\cdot, q) \qquad \text{(S-Stop)}$$
otherwise

Note that although rule S-Stop is not strictly necessary (rather than halting, the automaton could suppress all further actions), using S-Stop simplifies the automaton's specification and decreases its running time.

As before, we extend the single-step semantics to a multi-step semantics using the reflexivity and transitivity rules from Section 2.3.

4.2 Precisely Enforceable Properties

Similarly to truncation automata, suppression automata can precisely enforce any safety property and no other properties.

Theorem 3 (Precise S-Enforcement) *A property $\hat{P}$ on a system with action set $\mathcal{A}$ can be precisely enforced by some suppression automaton if and only if $\forall \sigma \in \mathcal{A}^\star$. $\neg\hat{P}(\sigma) \Rightarrow \forall \tau \in \mathcal{A}^\star . \neg\hat{P}(\sigma; \tau)$.*

Proof Both directions proceed completely analogously to the proof of Precise T-Enforcement given in Appendix A. □

4.3 Effectively Enforceable Properties

Suppression automata can effectively enforce a proper superset of the properties enforceable by truncation automata. We illustrate this with an example.

Consider an authenticated-login policy that requires users who wish to login to use an authenticated login, on a system with actions for both an unauthenticated login (ulogin) and an authenticated login (alogin). On this system, an unauthenticated login followed by an authenticated login is semantically equivalent to a single authenticated login (i.e., $\mathsf{ulogin}; \mathsf{alogin} \cong \mathsf{alogin}$); however, logging in more than once with an authenticated login is not equivalent to a single authenticated login (i.e., $\mathsf{alogin}; \mathsf{alogin} \ncong \mathsf{alogin}$).

A truncation automaton cannot effectively enforce this property because upon seeing a ulogin it can take no valid action. It cannot accept because an authenticated login may never occur, which violates the property. It also cannot halt because the next action may be alogin, making the automaton's input ($\mathsf{ulogin}; \mathsf{alogin}$) obey the property but not be equivalent to its output ($\cdot$). In contrast, a suppression automaton can effectively enforce this property simply by suppressing the ulogin. If the next action is alogin, then the suppression automaton accepts this and will have correctly effectively enforced the property since its output (alogin) is equivalent to its input ($\mathsf{ulogin}; \mathsf{alogin}$). If the alogin does not appear (and hence the input sequence violates the property), then the suppression automaton will not have emitted anything, thereby satisfying the property.

To formally characterize the properties that can be effectively enforced by suppression automata, we again generalize our automata to functions that act over sequences of symbols. Given a set of actions $\mathcal{A}$, a function $\omega^\star : \mathcal{A}^\star \to \mathcal{A}^\star$ is a *suppression-rewrite function* if it satisfies the following conditions.

1. $\omega^\star(\cdot) = \cdot$
2. $\forall \sigma \in \mathcal{A}^\star\ \forall a \in \mathcal{A}$.
   $\omega^\star(\sigma; a) = \omega^\star(\sigma); a$, or
   $\omega^\star(\sigma; a) = \omega^\star(\sigma)$

These conditions encode the restrictions placed on suppression automata that (1) nothing can be output when no actions are input and (2) an input action can only be accepted or suppressed. Halting is equivalent to suppressing all additional input actions.

The following theorem formally specifies the properties effectively enforceable by suppression automata.

Theorem 4 (Effective S-Enforcement) *A property $\hat{P}$ on a system with action set $\mathcal{A}$ can be effectively enforced by some suppression automaton if and only if there exists a computable suppression-rewrite function $\omega^\star$ such that for all executions $\sigma \in \mathcal{A}^\star$,*

1. *If $\neg\hat{P}(\sigma)$ then $\hat{P}(\omega^\star(\sigma))$*
2. *If $\hat{P}(\sigma)$ then $\sigma \cong \omega^\star(\sigma)$*

Proof See Appendix A. □

The essence of suppression automata as effective enforcers is that they can suppress an action if it is potentially bad and at the same time unnecessary. If the input sequence turns out to be in the property, then because the suppressed action was unnecessary, no harm was done.

5 Insertion Automata

This section introduces *insertion automata*, which have the power to insert actions into the action stream or halt the target program but cannot suppress actions. As with truncation and suppression automata, we define them formally and then consider what properties they can enforce precisely and effectively.

5.1 Definition

An *insertion automaton* I is a finite or countably infinite state machine (Q, q_0, δ, γ) that is defined with respect to some system with action set $\mathcal{A}$. The partial function $\delta : \mathcal{A} \times Q \to Q$ specifies the transition function as before. The new element is a partial function γ that specifies the insertion of a finite sequence of actions into the program's action sequence. We call this the *insertion function* and it has type $\mathcal{A} \times Q \to \overrightarrow{\mathcal{A}} \times Q$, where the first component in the returned pair indicates the finite and non-empty sequence of actions to be inserted. In order to maintain the determinacy of the automaton, we require the domain of the insertion function to be disjoint from the domain of the transition function.

We specify the execution of an insertion automaton as before. The single-step relation is defined below.

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

$$(\sigma, q) \xrightarrow{a}_I (\sigma', q') \qquad \text{(I-STEP)}$$
if $\sigma = a; \sigma'$
and $\delta(a, q) = q'$

$$(\sigma, q) \xrightarrow{\tau}_I (\sigma, q') \qquad \text{(I-INS)}$$
if $\sigma = a; \sigma'$
and $\gamma(a, q) = \tau, q'$

$$(\sigma, q) \xrightarrow{\cdot}_I (\cdot, q) \qquad \text{(I-STOP)}$$
otherwise

We also extend this single-step semantics to a multi-step semantics as before.

5.2 Precisely Enforceable Properties

Identically to truncation and suppression automata, insertion automata precisely enforce exactly the set of safety properties. Perhaps surprisingly, the power to suppress or insert actions does not enable security automata to precisely enforce more properties. This is because every illegal sequence must be transformed into a legal sequence, but doing so prevents any legal extension from being precisely enforced (because the sequence has already been modified and cannot be accepted in an entirely lockstep manner). Therefore, regardless of the transformational capabilities a security automaton has on execution sequences, it can only precisely enforce safety properties.

Theorem 5 (Precise I-Enforcement) *A property $\hat{P}$ on a system with action set $\mathcal{A}$ can be precisely enforced by some insertion automaton if and only if $\forall \sigma \in \mathcal{A}^\star . \neg\hat{P}(\sigma) \Rightarrow \forall \tau \in \mathcal{A}^\star . \neg\hat{P}(\sigma; \tau)$.*

Proof Both directions proceed completely analogously to the proof of Precise T-Enforcement given in Appendix A. □

5.3 Effectively Enforceable Properties

Let us now consider insertion automata operating as effective, rather than precise, enforcers. As with the other automata seen so far, insertion automata viewed in this light can enforce a greater range of properties. We illustrate this with a very application-specific example: the San Francisco cable car.

The property we want to enforce is that if a person boards the car (board), she must show her ticket either to the conductor (show_conductor) or to the driver (show_driver). She may show them her ticket either before or after getting onto the car, and it is OK, though redundant, to show the ticket twice. A person may board the car at most once. The only sequences not in the property are getting onto the car without ever showing a ticket and boarding more than once.

An insertion automaton can effectively enforce this property because upon seeing a bad sequence where no ticket has been shown but a passenger boards (board), it can insert the action show_driver. If the stream turns out to already contain either show_driver or show_conductor, no harm has been done (because show_driver; board; show_driver $\cong$ board; show_driver and show_driver; board; show_conductor $\cong$ board; show_conductor).

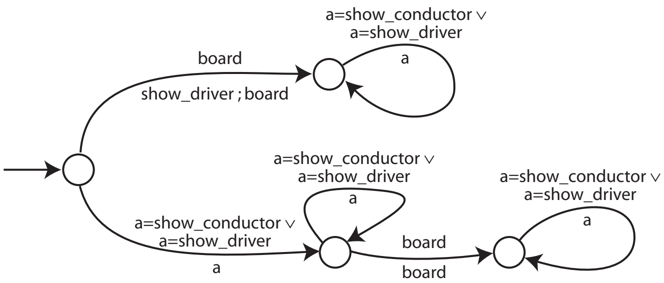

Fig. 1 An insertion automaton that effectively enforces the cable-car policy.

Figure 1 shows an insertion automaton that effectively enforces the cable-car policy. The nodes in the picture represent the automaton states and the arcs represent the transitions. The action above an arc triggers the transition, and the sequence below an arc represents the actions that are emitted. An arc with multiple symbols below it is an insertion transition. All other transitions are accepting transitions, and if there is no arc for the current action then the automaton halts.

A truncation automaton cannot effectively enforce this property because it has no way to handle an initial board action. If it accepts and there are no further actions, then it has allowed a passenger to board without showing a ticket; if it halts and the sequence does obey the property, then it has not preserved the semantics of the input sequence. Either way, it cannot effectively enforce the property.

As with precise enforcement, we can compare the powers of insertion and suppression automata as effective enforcers. The authenticated-login policy of Section 4.3 cannot be effectively enforced by an insertion automaton. When confronted with the sequence ulogin, an insertion automaton cannot accept because if there is no further input, the output would not obey the property; it cannot halt because the sequence could potentially be in the property (and $\mathsf{ulogin}; \mathsf{alogin} \ncong \cdot$); and it cannot insert an alogin because possible future alogin actions would make the stream obey the property yet not be equivalent to the stream with the inserted alogin.

This shows that there are properties that are effectively enforceable by suppression automata that are not effectively enforceable by insertion automata. Conversely, not all properties effectively enforceable by insertion automata can be effectively enforced by suppression automata. The cable-car example, for instance, cannot be effectively enforced by a suppression automaton. When confronted with a board action, a suppression automaton cannot accept because $\neg\hat{P}(\mathsf{board})$, and cannot halt or suppress because that could change a legal sequence into one that is not equivalent to it (e.g., $\mathsf{board}; \mathsf{show\_driver} \ncong \mathsf{show\_driver}$ and $\mathsf{board}; \mathsf{show\_driver} \ncong \cdot$).

Before we characterize the properties that can be effectively enforced by insertion automata, we again generalize our automata into functions that act over sequences of symbols. Given a set of actions $\mathcal{A}$, a function $\gamma^\star : \mathcal{A}^\star \rightarrow \mathcal{A}^\star$ is an *insertion-rewrite function* if it satisfies the following conditions.

1. $\gamma^\star(\cdot) = \cdot$
2. $\forall \sigma \in \mathcal{A}^\star\ \forall a \in \mathcal{A}\ \exists \tau \in \mathcal{A}^\star$.
   $\gamma^\star(\sigma; a) = \gamma^\star(\sigma); \tau; a$, or
   $\gamma^\star(\sigma; a) = \gamma^\star(\sigma); \tau \wedge (\forall \tau' \in \mathcal{A}^\star . \tau \neq \tau'; a) \wedge H(\gamma^\star, \sigma; a)$

where $H(\gamma^\star, \sigma)$ is the predicate defined in Section 3.3.

The intuition for insertion-rewrite functions is similar to that for the other rewrite functions, except that there are more possible behaviors of the corresponding automata to consider. When processing an action a, an insertion automaton can react in any of the following ways: it may accept a, insert some number of actions (the number of inserted actions is finite in this article because we limit our analysis to finite-length sequences) before accepting a, halt the target, or insert some number of actions before halting the target. The first clause of condition (2) above captures the first two of these possibilities by allowing the rewrite function to insert some number (possibly zero) of actions before accepting. The second part of condition (2) captures the other possibilities where some number (possibly zero) of actions are inserted before halting. For the sake of determinacy—that is, to make sure that only one of the two clauses in condition (2) applies—the case in which the automaton inserts some sequence that ends with a and then halts is handled as if a were accepted before halting. Note that by the definition of effective enforcement, insertion automata that effectively enforce properties only insert finitely many symbols during a run of a program. Therefore, the case in which an automaton does not eventually invoke operational rules I-STEP or I-STOP after some finite number of uses of rule I-INS does not apply.

The following theorem formally specifies the properties effectively enforceable by insertion automata.

Theorem 6 (Effective I-Enforcement) *A property $\hat{P}$ on a system with action set $\mathcal{A}$ can be effectively enforced by some insertion automaton if and only if there exists a computable insertion-rewrite function $\gamma^\star$ such that for all executions $\sigma \in \mathcal{A}^\star$,*

1. If $\neg\hat{P}(\sigma)$ then $\hat{P}(\gamma^\star(\sigma))$
2. If $\hat{P}(\sigma)$ then $\sigma \cong \gamma^\star(\sigma)$

Proof See Appendix A. □

The defining feature of insertion automata as effective enforcers is that they allow required actions to be inserted into a sequence whenever it is OK for the inserted actions to be repeated. If the automaton's input actually does obey the property, the automaton's output remains semantically equivalent to the input.

6 Edit Automata

We now turn to a far more powerful security automaton, the *edit automaton*. After defining edit automata formally and considering the properties they can enforce precisely and effectively, we present an extended example of an edit automaton that effectively enforces a market transaction policy.

6.1 Definition

We form an *edit automaton* E by combining the insertion automaton with the suppression automaton. Our machine is now described by a 5-tuple of the form $(Q, q_0, \delta, \gamma, \omega)$. The operational semantics is derived from the composition of the operational rules of the two previous automata. Again, the partial functions δ and ω have the same domain while δ and γ have disjoint domains.

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

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

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

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

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

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

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

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

otherwise

Although rule E-StepA is not strictly necessary (rather than accepting an action a, the automaton could insert a and then suppress the original a), its presence allows acceptance of an action in only one step. This simplifies the specification of the automaton and in general decreases its running time. Similarly, and as with suppression automata, the effect of rule E-Stop can be accomplished by suppressing any further input, again at the cost of increased running time and specification complexity.

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

6.2 Precisely Enforceable Properties

Edit automata precisely enforce exactly the set of safety properties. This follows immediately from the discussion in Section 5.2—as precise enforcers, edit automata have the same power as truncation, suppression, and insertion automata.

Theorem 7 (Precise E-Enforcement) *A property $\hat{P}$ on a system with action set $\mathcal{A}$ can be precisely enforced by some edit automaton if and only if $\forall \sigma \in \mathcal{A}^\star . \neg\hat{P}(\sigma) \Rightarrow \forall \tau \in \mathcal{A}^\star . \neg\hat{P}(\sigma; \tau)$.*

Proof Both directions proceed completely analogously to the proof of Precise T-Enforcement given in Appendix A. □

6.3 Effectively Enforceable Properties

Similarly to truncation, suppression, and insertion automata, considering edit automata as effective enforcers enables them to enforce a wider range of properties. In addition, edit automata can effectively enforce properties effectively enforceable by neither suppression nor insertion automata. Section 6.4 illustrates this with a detailed example.

Edit automata are very powerful effective enforcers because they can insert actions that are required and later suppress those same actions if they appear in the original input sequence. Conversely, an edit automaton can suppress a sequence of potentially illegal actions, and if the sequence is later determined to be legal, just re-insert it. Interestingly, with this technique edit automata can effectively enforce any property—the automaton simply suppresses all actions until it can confirm that the current prefix obeys the property, at which point it inserts all the suppressed actions. Any legal input will thus be output without modification. If an input is illegal, an edit automaton will output its longest valid prefix.

The following theorem formalizes this idea, and its proof shows how to construct an edit automaton to effectively enforce any property. This theorem holds on all systems because it only uses strict equality as the equivalence relation. However, as with all theorems presented in this article, the following theorem assumes that action sequences have finite length. Without this assumption, many interesting properties, such as the one requiring that all executions terminate, cannot be effectively enforced by edit automata.

Theorem 8 (Effective E-Enforcement) *Any property $\hat{P}$ on a system with action set $\mathcal{A}$ can be effectively enforced by some edit automaton.*

Proof See Appendix A. □

6.4 An Example: Transactions

To demonstrate the power of our edit automata, we show how to implement the monitoring of transactions. The desired properties of atomic transactions [EN94], commonly referred to as the ACID properties, are atomicity (either the entire transaction is executed or no part of it is executed), 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).

The first property, atomicity, can be modeled using an edit automaton by creating an intentions log. That is, the automaton suppresses input actions from the start of the transaction and, if the transaction completes successfully, the entire sequence of actions is emitted atomically to the output stream; otherwise it is discarded. Consistency preservation can be enforced by simply verifying that the sequence to be emitted leaves the system in a consistent state. The durability or permanence of a committed transaction is ensured by the fact that committing a transaction is modeled by emitting the corresponding sequence of actions to the output stream. Once an action has been written to the output stream it can no longer be touched by the automaton; furthermore, failed transactions output nothing. We only model the actions of a single agent in this example and therefore ignore issues of isolation.

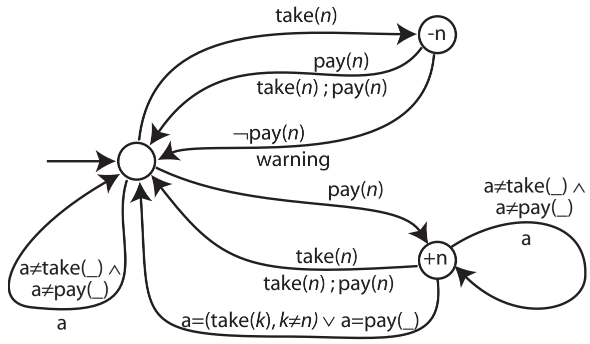

Fig. 2 An edit automaton that effectively enforces the market policy.

To make our example more concrete, we will model a simple market system with two main actions, take(n) and pay(n), which represent acquisition of n apples and the corresponding payment. We let a range over all the actions that might occur in the system (such as `take`, `pay`, `window-shop`, `browse`, etc.). Our policy is that every time an agent takes n apples it must pay for those apples. Payments may come before acquisition or vice versa, and take(n); pay(n) is semantically equivalent to pay(n); take(n). The automaton enforces the atomicity of this transaction by emitting take(n); pay(n) only when the transaction completes. If payment is made first, the automaton allows clients to perform other actions such as `browse` before paying (the take-pay transaction appears atomically after all such intermediary actions). On the other hand, if apples are taken and not paid for immediately, we issue a warning and abort the transaction. Consistency is ensured by remembering the number of apples taken or the size of the prepayment in the state of the machine. Once acquisition and payment occur, the sale is final and there are no refunds (durability). Figure 2 displays the infinite-state edit automaton that effectively enforces our market policy. Arcs with no symbols beneath them represent suppression transitions.

Neither an insertion automation nor a suppression automaton can effectively enforce the market policy. An insertion automaton, if it encounters pay(n), can take no action: it cannot halt because the next action may be take(n), it cannot accept because there may be no more actions, and it cannot insert take(n) because that may disrupt a valid sequence of take(n) and pay(n) in an input that obeys the property. A suppression automaton, upon encountering take(n), can take no action: it cannot suppress or halt because the next action may be pay(n), and it cannot accept because there may be no more actions.

7 Related Work

Schneider [Sch00] was the first to analyze the power of security mechanisms. He defined the set of security properties precisely enforceable by truncation automata and observed that it was a subset of the safety properties. Schneider also briefly mentions mechanisms more powerful than truncation automata, but he does not give a detailed analysis of their power. Schneider's definition provides an upper bound on the set of properties that can be precisely enforced by security automata. However, it is a loose upper bound. Viswanathan, Kim, and others [Vis00, KKL+02] demonstrate that to obtain a tighter bound on the power of run-time monitors, one must add computability constraints to the definition of truncation automata. Viswanathan [Vis00] has also demonstrated that the set of properties enforceable through run-time monitoring is equivalent to the CoRE properties. Fong [Fon04] formally demonstrates that limiting the amount of run-time information that a monitor may access limits the monitor's power; further, Fong develops a classification of run-time monitors according to this metric.

Concurrently with our work, Hamlen, Morrisett, and Schneider [HMS03] have begun to investigate the power of a broader set of enforcement mechanisms. They consider security mechanisms based upon static analysis, run-time monitoring, and program rewriting and compare and contrast the power of these mechanisms. They observe that the statically enforceable properties correspond to the recursively decidable properties of programs. Taken with Viswanathan's result, this fact implies that run-time monitors (when given access to the source program text) can enforce strictly more properties than can be enforced through static analysis. Hamlen et al. also prove that program rewriters do not correspond to any complexity class in the arithmetic hierarchy.

In contrast with these other theoretical research efforts, our work provides a detailed analysis of the power of run-time monitors. We introduce several kinds of monitors, each with different run-time capabilities (i.e., truncation, suppression, and insertion). We clearly specify what it means to enforce a property, and we demonstrate that the power of these monitors varies depending upon the context in which they are used.

Implementation efforts [ET99, UES99, UES00, KVBA+99, BLW04] are considerably more advanced than the corresponding theoretical investigations. In general, these systems allow arbitrary code to be executed in response to a potential security violation, so edit automata provide a reasonable model for attempting to understand their behavior. In most cases, these languages can be considered domain-specific aspect-oriented programming languages [KHH+01], which have been designed to facilitate the enforcement of security properties.

Other researchers have investigated optimization techniques for run-time monitors [CF00, Thi01] and certification of programs instrumented with security checks [Wal00]. Kim et al. [KKL+02] analyze the performance cost of run-time monitors and show that the language one uses to define monitors has a significant impact on performance. They also define optimizations to decrease the overhead of run-time monitoring.

Run-time monitoring and checking can also be used in settings where security is not necessarily the primary focus. Lee et al. [KVBA+99], for example, have developed a monitoring system specifically for improving the reliability of real-time systems. Sandholm and Schwartzbach have used run-time monitoring to ensure that concurrently executing Web scripts obey important safety properties [SS98].

8 Conclusions

In this article, we have introduced a detailed framework for reasoning about the power of security mechanisms that are able to intercept and modify untrusted program actions at run time.

Before we begin the analysis proper, we carefully define what it means to enforce a property at run time. Our definitions of policy enforcement are motivated by two main considerations:

(Soundness) The final output of a monitored system must obey the policy. Consequently, bad programs that would otherwise violate the policy must have their executions modified by the enforcement mechanism.

(Transparency) Whenever the untrusted program obeys the policy in question, a run-time enforcement mechanism should preserve the semantics of the untrusted program. In other words, the actions of the enforcement mechanism should not be observable when monitoring good programs that do not violate the policy.

One obtains slightly different definitions of enforcement depending upon the interpretation of what it means for a mechanism to be "semantics-preserving." Consequently, it is necessary to ensure that the definition of enforcement is appropriate for the application and context at hand.

Once the definition of enforcement is set, we investigate the power of a hierarchy of monitors, each with varying run-time capabilities, including truncation, suppression, and insertion of security-relevant program actions. We demonstrate that the power of these different kinds of monitors varies depending upon which definition of enforcement is applied. Under the strictest interpreation of (Transparency), all security automata enforce exactly the set of safety properties. Under a more relaxed interpretation of (Transparency), we develop the hierarchy shown in Figure 3.

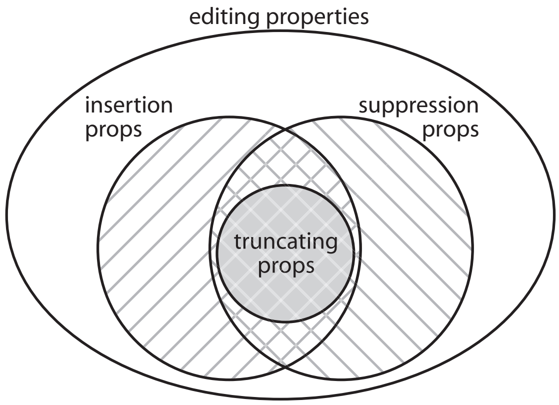

Fig. 3 A taxonomy of *effectively* enforceable security properties.

This article sets some of the fundamental limits of security mechanisms that operate by monitoring and modifying program behavior at run time. However, there are many open questions that continue to spark our interest in this area. In particular, we desire to understand further the impact of constraining the resources available either to the running program or the run-time monitor. For example, are there practical properties that can be enforced by exponential-time but not polynomial-time monitors? What effect does bounding the space available to edit automata have on the set of enforceable properties? What if we limit the program's access to random bits and therefore its ability to use strong cryptography? Can we generalize our analyses to infinite sequences? These unanswered questions and many similar ones suggest a variety of new research directions in this nascent research field.

Acknowledgements Enlightening discussions with Kevin Hamlen, Greg Morrisett, and Fred Schneider helped to stimulate our research and improve this article. We would also like to thank the anonymous reviewers for their helpful and thorough comments.

References

[AS87] Bowen Alpern and Fred Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1987.

[BLW02] Lujo Bauer, Jarred Ligatti, and David Walker. More enforceable security policies. In *Foundations of Computer Security*, Copenhagen, Denmark, July 2002.

[BLW04] Lujo Bauer, Jay Ligatti, and David Walker. A language and system for enforcing run-time security policies. Technical Report TR-699-04, Princeton University, January 2004.

[CF00] Thomas Colcombet and Pascal Fradet. Enforcing trace properties by program transformation. In *Twenty-Seventh ACM Symposium on Principles of Programming Languages*, pages 54–66, Boston, January 2000. ACM Press.

[EN94] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of database systems.* The Benjamin/Cummings Publishing Company, Inc., 1994.

[ET99] David Evans and Andrew Twyman. Flexible policy-directed code safety. In *IEEE Security and Privacy*, Oakland, CA, May 1999.

[Fon04] Philip W. L. Fong. Access control by tracking shallow execution history. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, Oakland, California, USA, May 2004.

[HMS03] Kevin Hamlen, Greg Morrisett, and Fred Schneider. Computability classes for enforcement mechanisms. Technical Report TR2003-1908, Cornell University, 2003.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. An overview of AspectJ. In *European Conference on Object-oriented Programming*. Springer-Verlag, 2001.

[KKL+02] Moonjoo Kim, Sampath Kannan, Insup Lee, Oleg Sokolsky, and Mahesh Viswantathan. Computational analysis of run-time monitoring—fundamentals of Java-MaC. In *Run-time Verification*, June 2002.

[KVBA+99] Moonjoo Kim, Mahesh Viswanathan, Hanene Ben-Abdallah, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Formally specified monitoring of temporal properties. In *European Conference on Real-time Systems*, York, UK, June 1999.

[Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions of Software Engineering*, 3(2):125–143, 1977.

[Sch00] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and Systems Security*, 3(1):30–50, February 2000.

[SS98] Anders Sandholm and Michael Schwartzbach. Distributed safety controllers for web services. In *Fundamental Approaches to Software Engineering*, volume 1382 of *Lecture Notes in Computer Science*, pages 270–284. Springer-Verlag, 1998.

[Thi01] Peter Thiemann. Enforcing security properties by type specialization. In *European Symposium on Programming*, Genova, Italy, April 2001.

[UES99] Úlfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the New Security Paradigms Workshop*, pages 87–95, Caledon Hills, Canada, September 1999.

[UES00] Úlfar Erlingsson and Fred B. Schneider. IRM enforcement of Java stack inspection. In *IEEE Symposium on Security and Privacy*, pages 246–255, Oakland, California, May 2000.

[Vis00] Mahesh Viswanathan. *Foundations for the Run-time Analysis of Software Systems.* PhD thesis, University of Pennsylvania, 2000.

[Wal00] David Walker. A type system for expressive security policies. In *Twenty-Seventh ACM Symposium on Principles of Programming Languages*, pages 254–267, Boston, January 2000.

A Proofs of Theorems

Theorem 1 (Precise T-Enforcement) *A property $\hat{P}$ on a system with action set $\mathcal{A}$ can be precisely enforced by some truncation automaton if and only if $\forall \sigma \in \mathcal{A}^\star. \neg\hat{P}(\sigma) \Rightarrow \forall \tau \in \mathcal{A}^\star. \neg\hat{P}(\sigma;\tau)$.*

Proof *(If Direction)* We construct a truncation automaton that precisely enforces any such $\hat{P}$ as follows.

- States: $q \in \mathcal{A}^\star$ (the sequence of actions seen so far)
- Start state: $q_0 = \cdot$ (the empty sequence)
- Transition function (δ):
  Consider processing the action a in state σ.
  (A) If $\hat{P}(\sigma;a)$ then emit a and continue in state $\sigma;a$.
  (B) Otherwise, simply stop (i.e., we leave δ is undefined in this case).
  This transition function is (Turing Machine) computable because $\hat{P}$ must by definition be computable.

The automaton maintains the invariant $\mathrm{INV}_p(q)$ that $q = \sigma$, σ has been output, and $\hat{P}(\sigma)$. The automaton can initially establish $\mathrm{INV}_p(q_0)$ since our definition of a property assumes $\hat{P}(\cdot)$ for all properties. A simple inductive argument on the length of the input σ suffices to show that the invariant is maintained for all inputs.

We now show that this automaton precisely enforces $\hat{P}$ on any $\sigma \in \mathcal{A}^\star$. There are two cases to consider.

- **Case** $\hat{P}(\sigma)$:
  Consider any prefix σ' of σ. If $\neg\hat{P}(\sigma')$ then by the safety constraint stated in the theorem and the fact that $\sigma' \preceq \sigma$, we would have $\neg\hat{P}(\sigma)$, contrary to the assumption that $\hat{P}(\sigma)$. Therefore, $\hat{P}(\sigma')$ must be true for all prefixes σ' of σ, so by examination of the transition function given above, the automaton must accept every prefix of σ without ever halting.
- **Case** $\neg\hat{P}(\sigma)$:
  INV_p maintains that regardless of the state of the automaton, it has always emitted some σ such that $\hat{P}(\sigma)$.

The automaton correctly precisely enforces $\hat{P}$ in both cases.

(Only-If Direction) Consider any $\sigma \in \mathcal{A}^\star$ such that $\neg\hat{P}(\sigma)$. Because a truncation automaton T precisely enforces $\hat{P}$ and $\neg\hat{P}(\sigma)$, it cannot be the case that $(\sigma, q_0) \stackrel{\sigma}{\Longrightarrow}_T (\cdot, q')$ for some q'. Suppose for the sake of obtaining a contradiction that $\exists \tau \in \mathcal{A}^\star.\hat{P}(\sigma;\tau)$. Then by the definition of precise enforcement, all actions of $\sigma;\tau$ must be accepted without any editing, implying (since σ is later extended to $\sigma;\tau$) that $(\sigma, q_0) \stackrel{\sigma}{\Longrightarrow}_T (\cdot, q')$ for some q', which we just showed cannot be true. Therefore, $\exists \tau \in \mathcal{A}^\star.\hat{P}(\sigma;\tau)$ is not true, so $\forall \tau \in \mathcal{A}^\star.\neg\hat{P}(\sigma;\tau)$.

□

Theorem 2 (Effective T-Enforcement) *A property $\hat{P}$ on a system with action set $\mathcal{A}$ can be effectively enforced by some truncation automaton if and only if there exists a computable truncation-rewrite function $\alpha^\star$ such that for all executions $\sigma \in \mathcal{A}^\star$,*

1. *If $\neg\hat{P}(\sigma)$ then $\hat{P}(\alpha^\star(\sigma))$*
2. *If $\hat{P}(\sigma)$ then $\sigma \cong \alpha^\star(\sigma)$*

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

- States: $q \in \mathcal{A}^\star$ (the sequence of actions seen so far)
- Start state: $q_0 = \cdot$ (for the empty sequence)
- Transition function (δ):
  Consider processing the action a in state σ.
  (A) If $\alpha^\star(\sigma;a) = \alpha^\star(\sigma);a$ then emit a and continue in state $\sigma;a$.
  (B) Otherwise, $\alpha^\star(\sigma;a) = \alpha^\star(\sigma)$ and $H(\alpha^\star, \sigma;a)$. Halt in this case (i.e., leave δ undefined).
  This transition function is (Turing Machine) computable due to the assumption that $\alpha^\star$ is computable.

The automaton maintains the invariant $\textsc{Inv}_p(q)$ that $q = \sigma$, σ is the input so far, and the automaton has emitted $\alpha^\star(\sigma)$. Initially, $\textsc{Inv}_p(q_0)$ because $\alpha^\star(\cdot) = \cdot$, and a simple inductive argument on the length of the input σ suffices to show that the invariant is maintained for all inputs.

We now show that the automaton emits $\alpha^\star(\sigma)$ when the input is σ. There are two cases, derived by inspection of the truncation automaton. In the first case, the automaton halts in state σ, so by $\textsc{Inv}_p(\sigma)$, $\alpha^\star(\sigma)$ has been emitted. Otherwise, the automaton must halt in some state σ' such that $\sigma' \preceq \sigma$ and $H(\alpha^\star, \sigma')$. $\textsc{Inv}_p(\sigma')$ then implies that $\alpha^\star(\sigma')$ has been emitted when the automaton halts. However, $H(\alpha^\star, \sigma')$ and $\sigma' \preceq \sigma$ imply that $\alpha^\star(\sigma') = \alpha^\star(\sigma)$, so the automaton has in fact emitted $\alpha^\star(\sigma)$ when it halts in this case as well.

Consider any execution $\sigma \in \mathcal{A}^\star$. By clause (1) in the theorem statement, if $\neg\hat{P}(\sigma)$ then $\hat{P}(\alpha^\star(\sigma))$. Because $\alpha^\star(\sigma)$ is actually what the automaton outputs on input σ, it correctly effectively enforces any $\sigma \in \mathcal{A}^\star$ such that $\neg\hat{P}(\sigma)$. Similarly, clause (2) of the theorem states that if $\hat{P}(\sigma)$ then $\sigma \cong \alpha^\star(\sigma)$, so the automaton's output is equivalent to its input when $\hat{P}(\sigma)$. The automaton therefore effectively enforces $\hat{P}$.

(Only-If Direction) Define $\alpha^\star(\sigma)$ to be whatever sequence is emitted by the truncation automaton on input σ. By this definition, $\alpha^\star$ is a computable function because the automaton has a (Turing Machine) computable transition function and is an effective enforcer (which by definition halts on all inputs). We first show that this is a truncation-rewrite function. Clearly, $\alpha^\star(\cdot) = \cdot$. When processing some action a after having already processed any sequence σ, the automaton may step via T-Step or T-Stop.

- **Case T-Step**: The automaton emits whatever has been emitted in processing σ (by definition, this is $\alpha^\star(\sigma)$), followed by a.
- **Case T-Stop**: The automaton emits only what has already been emitted, so $\alpha^\star(\sigma;a) = \alpha^\star(\sigma)$. Because the automaton halts, it may not examine any remaining input, implying that $H(\alpha^\star, \sigma;a)$ is true.

In any case, the truncation automaton's output adheres to the definition of a truncation-rewrite function.

Now consider an arbitrary $\sigma \in \mathcal{A}^\star$. By the definition of effective enforcement and that $\alpha^\star(\sigma)$ is defined to be whatever sequence is emitted by the truncation automaton on input σ, $\hat{P}(\alpha^\star(\sigma))$ and $\hat{P}(\sigma) \Rightarrow \sigma \cong \alpha^\star(\sigma)$. □

Theorem 4 (Effective S-Enforcement) *A property $\hat{P}$ on a system with action set $\mathcal{A}$ can be effectively enforced by some suppression automaton if and only if there exists a computable suppression-rewrite function $\omega^\star$ such that for all executions $\sigma \in \mathcal{A}^\star$,*

1. *If $\neg\hat{P}(\sigma)$ then $\hat{P}(\omega^\star(\sigma))$*
2. *If $\hat{P}(\sigma)$ then $\sigma \cong \omega^\star(\sigma)$*

Proof *(If Direction)* We construct a suppression automaton that effectively enforces any such $\hat{P}$ as follows.

- States: $q \in pre(\Sigma)$ (the sequence of actions seen so far)
- Start state: $q_0 = \cdot$ (the empty sequence)
- Transition and suppression functions (combined for simplicity):
  Consider processing the action a in state $q = \sigma$.
  (A) If $\omega^\star(\sigma;a) = \omega^\star(\sigma);a$ then emit a and continue in state $\sigma;a$.
  (B) Otherwise, $\omega^\star(\sigma;a) = \omega^\star(\sigma)$. Suppress a and continue in state $\sigma;a$.
  This transition function is (Turing Machine) computable due to the assumption that $\omega^\star$ is computable.

This suppression automaton maintains the invariant $\textsc{Inv}_p(q)$ that $q = \sigma$, σ is the input seen so far, and $\omega^\star(\sigma)$ has been output. Initially, $\textsc{Inv}_p(q_0)$ because $\omega^\star(\cdot) = \cdot$, and a simple inductive argument on the length of the input σ suffices to show that the invariant is maintained for all inputs.

By inspection of the automaton, we note that when processing some input $\sigma \in \mathcal{A}^\star$, the automaton always halts in state σ because there is no more input to process.

Thus, $\textsc{Inv}_p(\sigma)$ ensures that the automaton always emits $\omega^\star(\sigma)$ whenever the input to the automaton is σ.

Analogously to the argument given in the Effective T-Enforcement theorem, because the automaton always outputs $\omega^\star(\sigma)$ on input σ, clauses (1) and (2) in the theorem statement ensure that the automaton effectively enforces $\hat{P}$.

(Only-If Direction) Define $\omega^\star(\sigma)$ to be whatever sequence is emitted by the suppression automaton on input σ. By this definition, $\omega^\star$ is a computable function because the automaton has a (Turing Machine) computable transition function and is an effective enforcer (which by definition halts on all inputs). We first show that this is indeed a suppression-rewrite function. Clearly, $\omega^\star(\cdot) = \cdot$. When processing some action a after having already processed any sequence σ, the automaton may step via S-StepA, S-StepS, or S-Stop. In the S-StepA case, the automaton emits whatever has been emitted in processing σ (by definition, this is $\omega^\star(\sigma)$), followed by a. In the other cases, the automaton emits only $\omega^\star(\sigma)$. Hence, $\omega^\star$ is a valid suppression function.

Now consider an arbitrary $\sigma \in \mathcal{A}^\star$. By the definition of effective enforcement and that $\omega^\star(\sigma)$ is defined to be whatever sequence is emitted by the suppression automaton on input σ, $\hat{P}(\omega^\star(\sigma))$ and $\hat{P}(\sigma) \Rightarrow \sigma \cong \omega^\star(\sigma)$, which implies clauses (1) and (2) of the theorem statement.

□

Theorem 6 (Effective I-Enforcement) *A property $\hat{P}$ on a system with action set $\mathcal{A}$ can be effectively enforced by some insertion automaton if and only if there exists a computable insertion-rewrite function $\gamma^\star$ such that for all executions $\sigma \in \mathcal{A}^\star$,*

1. *If $\neg\hat{P}(\sigma)$ then $\hat{P}(\gamma^\star(\sigma))$*
2. *If $\hat{P}(\sigma)$ then $\sigma \cong \gamma^\star(\sigma)$*

Proof *(If Direction)* We construct an insertion automaton that effectively enforces any such $\hat{P}$ as follows.

– States: $q \in (\mathcal{A}^\star \times \{+,-\}) \cup \{\mathsf{end}\}$ (the sequence of actions seen so far paired with + (-) to indicate that the automaton did not (did) most recently step via rule I-Ins, or end if the automaton will stop on the next step)
– Start state: $q_0 = \langle\cdot,+\rangle$ (for the empty sequence)
– Transition and insertion functions (for simplicity, we combine δ and γ):
Consider processing the action a in state q.
(A) If $q = \mathsf{end}$ then stop.
(B) If $q = \langle\sigma;a,-\rangle$ then emit (accept) a and continue in state $\langle\sigma;a,+\rangle$.
(C) If $q = \langle\sigma,+\rangle$ and $\gamma^\star(\sigma;a) = \gamma^\star(\sigma);\tau;a$ then insert τ and continue in state $\langle\sigma;a,-\rangle$.
(D) Otherwise, $q = \langle\sigma,+\rangle$, $\gamma^\star(\sigma;a) = \gamma^\star(\sigma);\tau$, $\forall\tau' \in \mathcal{A}^\star.\tau \neq \tau';a$, and $H(\gamma^\star,\sigma;a)$. Insert τ and continue in state end.

This transition function is (Turing Machine) computable due to the assumption that $\gamma^\star$ is computable.

The automaton maintains the following invariant $\textsc{Inv}_p(q)$.

– If $q = \langle\sigma,+\rangle$ then σ is the input so far and the automaton has emitted $\gamma^\star(\sigma)$.
– If $q = \mathsf{end}$ then σ is the input so far, the automaton has emitted $\gamma^\star(\sigma)$, and $H(\gamma^\star,\sigma)$.
– Otherwise, $q = \langle\sigma;a,-\rangle$, $\sigma;a$ is the input so far, $\gamma^\star(\sigma;a) = \gamma^\star(\sigma);\tau;a$, and the automaton has emitted $\gamma^\star(\sigma);\tau$.

Initially, $\textsc{Inv}_p(q_0)$ because $\gamma^\star(\cdot) = \cdot$, and a simple inductive argument on the length of the input σ suffices to show that the invariant is maintained for all inputs.

We now show that the automaton emits $\gamma^\star(\sigma)$ when the input is σ. There are two cases, derived by inspection of the insertion automaton. In the first case, the automaton halts in state $\langle\sigma,+\rangle$, so by $\textsc{Inv}_p(\sigma)$, $\gamma^\star(\sigma)$ has been emitted. Otherwise, the automaton must halt in state end, having only seen input σ' such that $\sigma' \preceq \sigma$. In this case, $\textsc{Inv}_p(\mathsf{end})$ implies that $\gamma^\star(\sigma')$ has been emitted when the automaton halts and $H(\gamma^\star,\sigma')$. However, $H(\gamma^\star,\sigma')$ and $\sigma' \preceq \sigma$ imply that $\gamma^\star(\sigma') = \gamma^\star(\sigma)$, so the automaton has in fact emitted $\gamma^\star(\sigma)$ when it halts in this case as well. Note that the automaton cannot halt in state $\langle\sigma;a,-\rangle$ because the next step will always be to emit a and continue in state $\langle\sigma;a,+\rangle$ (using transition (B)).

Analogously to the argument given in the Effective T-Enforcement theorem, because the automaton always outputs $\gamma^\star(\sigma)$ on input σ, clauses (1) and (2) in the theorem statement ensure that the automaton effectively enforces $\hat{P}$.

(Only-If Direction) Define $\gamma^\star(\sigma)$ to be whatever sequence is emitted by the insertion automaton on input σ. By this definition, $\gamma^\star$ is a computable function because the automaton has a (Turing Machine) computable transition function and is an effective enforcer (which by definition halts on all inputs). We first show that this is an insertion-rewrite function. Clearly, $\gamma^\star(\cdot) = \cdot$. When processing some action a after having already processed any sequence σ, the automaton may step via I-Step, I-Ins, or I-Stop.

– **Case I-Step**: Here the automaton emits whatever has been emitted in processing σ (by definition, this is $\gamma^\star(\sigma)$), followed by a. Thus, $\gamma^\star(\sigma;a) = \gamma^\star(\sigma);\tau;a$, where $\tau = \cdot$.
– **Case I-Stop**: Here the automaton emits only what has already been emitted, so $\gamma^\star(\sigma;a) = \gamma^\star(\sigma);\tau$, where $\tau = \cdot$ and $\forall\tau' \in \mathcal{A}^\star.\tau \neq \tau';a$. Because the automaton halts, it may not examine any remaining input, implying that $H(\gamma^\star,\sigma;a)$.
– **Case I-Ins**: Because the automaton effectively enforces $\hat{P}$, it may only insert a finite sequence of symbols before either accepting or halting on the current

action. That is, although the automaton may apply rule I-INS multiple times in succession, the end result must be the insertion of only a finite sequence of actions, followed by acceptance or termination. In the case of ultimate acceptance, the net output of the automaton, $\gamma^\star(\sigma;a)$, is $\gamma^\star(\sigma);\tau;a$ for some $\tau \in \mathcal{A}^\star$. In the case of ultimate termination, no more input may be examined, so $H(\gamma^\star,\sigma;a)$ and for some $\tau \in \mathcal{A}^\star$, either $\gamma^\star(\sigma;a) = \gamma^\star(\sigma);\tau;a$ (if a was the last inserted action), or $\gamma^\star(\sigma;a) = \gamma^\star(\sigma);\tau$ and $\forall \tau' \in \mathcal{A}^\star.\tau \neq \tau';a$ (if a was not the last inserted action).

In any case, the insertion automaton's output adheres to the definition of an insertion-rewrite function.

Now consider an arbitrary $\sigma \in \mathcal{A}^\star$. By the definition of effective enforcement and that $\gamma^\star(\sigma)$ is defined to be whatever sequence is emitted by the insertion automaton on input σ, $\hat{P}(\gamma^\star(\sigma))$ and $\hat{P}(\sigma) \Rightarrow \sigma \cong \gamma^\star(\sigma)$, which implies clauses (1) and (2) of the theorem statement. □

Theorem 8 (Effective E-Enforcement) *Any property $\hat{P}$ on a system with action set $\mathcal{A}$ can be effectively enforced by some edit automaton.*

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

- States: $q \in \mathcal{A}^\star \times \mathcal{A}^\star \times \{+,-\}$ (the sequence of actions seen so far, the actions seen but not emitted, and + (-) to indicate that the automaton must not (must) suppress the current action)
- Start state: $q_0 = \langle \cdot, \cdot, + \rangle$ (for the empty sequence)
- Transition, insertion, and suppression functions (for simplicity, we combine δ, γ, and ω):
  Consider processing the action a in state q.
  (A) If $q = \langle \sigma, \tau, + \rangle$ and $\neg\hat{P}(\sigma;a)$ then suppress a and continue in state $\langle \sigma;a, \tau;a, + \rangle$.
  (B) If $q = \langle \sigma, \tau, + \rangle$ and $\hat{P}(\sigma;a)$ then insert $\tau;a$ and continue in state $\langle \sigma;a, \cdot, - \rangle$.
  (C) Otherwise, $q = \langle \sigma;a, \cdot, - \rangle$. Suppress a and continue in state $\langle \sigma;a, \cdot, + \rangle$.
  This transition function is (Turing Machine) computable because $\hat{P}$ must by definition be computable.

The automaton maintains the following invariant $\text{INV}_p(q)$.

- If $q = \langle \sigma, \tau, + \rangle$ then σ is the input so far, $\sigma = \sigma';\tau$ (for some $\sigma' \in \mathcal{A}^\star$), the automaton has emitted σ', and σ' is the longest prefix of σ such that $\hat{P}(\sigma')$.
- Otherwise, $q = \langle \sigma;a, \cdot, - \rangle$, $\sigma;a$ is the input so far, $\sigma;a$ has been emitted, and $\hat{P}(\sigma;a)$.

Initially, $\text{INV}_p(q_0)$ because $\hat{P}(\cdot)$, and a simple inductive argument on the length of the input σ suffices to show that the invariant is maintained for all inputs.

By inspection, we note that the edit automaton cannot halt in state $\langle \sigma;a, \cdot, - \rangle$ because the next step will always be to suppress a and continue in state $\langle \sigma;a, \cdot, + \rangle$ (using transition (C)). Therefore, the automaton always halts in some state $\langle \sigma, \tau, + \rangle$ on input σ, so $\text{INV}_p(\langle \sigma, \tau, + \rangle)$ ensures that the automaton emits some σ' such that $\hat{P}(\sigma')$ on all inputs. In addition, $\text{INV}_p(\langle \sigma, \tau, + \rangle)$ implies that if $\hat{P}(\sigma)$ then $\sigma = \sigma'$, so the automaton must emit exactly σ on input σ whenever $\hat{P}(\sigma)$. This edit automaton thus effectively enforces $\hat{P}$. □