

More Enforceable Security Policies*

Lujo Bauer, Jarred Ligatti and David Walker
Department of Computer Science
Princeton University
Princeton, NJ 08544

Abstract

We analyze the space of security policies that can be enforced by monitoring programs at runtime. Our program monitors are automata that examine the sequence of program actions and transform the sequence when it deviates from the specified policy. The simplest such automaton truncates the action sequence by terminating a program. Such automata are commonly known as security automata, and they enforce Schneider's EM class of security policies. We define automata with more powerful transformational abilities, including the ability to insert a sequence of actions into the event stream and to suppress actions in the event stream without terminating the program. We give a set-theoretic characterization of the policies these new automata are able to enforce and show that they are a superset of the EM policies.

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 and should we demand of our system?
2. What mechanisms should we implement 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.

Recently, Schneider [Sch00] attacked this question by defining EM, a subset of safety properties [Lam85, AS87] that has a general-purpose enforcement mechanism - a *security automaton* that interposes itself between the program and the machine on which the

program runs. It examines the sequence of security-relevant program actions one at a time and if the automaton recognizes an action that will violate its policy, it terminates the program. The mechanism is very general since decisions about whether or not to terminate the program can depend upon the entire history of the program execution. However, since the automaton is only able to recognize bad sequences of actions and then terminate the program, it can only enforce safety properties.

In this paper, we re-examine the question of which security policies can be enforced at runtime by monitoring program actions. Following Schneider, we use automata theory as the basis for our analysis of enforceable security policies. However, we take the novel approach that these automata are transformers on the program action stream, rather than simple recognizers. This viewpoint leads us to define two new enforcement mechanisms: an *insertion automaton* that is able to insert a sequence of actions into the program action stream, and a *suppression automaton* that suppresses certain program actions rather than terminating the program outright. When joined, the insertion automaton and suppression automaton become an *edit automaton*. We characterize the class of security policies that can be enforced by each sort of automaton and provide examples of important security policies that lie in the new classes and outside the class EM.

Schneider is cognizant that the power of his automata is limited by the fact that they can only terminate programs and may not modify them. However, to the best of our knowledge, neither he nor anyone else has formally investigated the power of a broader class of runtime enforcement mechanisms that explicitly manipulate the program action stream. Erlingsson and Schneider [UES99] have implemented inline reference monitors, which allow arbitrary code to be executed in response to a violation of the security policy, and have demonstrated their effectiveness on a range of security policies of different levels of abstraction from the Software Fault Isolation policy for the Pentium IA32 architecture to the Java stack in-

*Appeared in *Foundations of Computer Security Workshop* (affiliated with *LICS '02*), Copenhagen, Denmark, July 2002.

spection policy for Sun’s JVM [UES00]. Evans and Twyman [ET99] have implemented a very general enforcement mechanism for Java that allows system designers to write arbitrary code to enforce security policies. Such mechanisms may be more powerful than those that we propose here; these mechanisms, however, have no formal semantics, and there has been no analysis of the classes of policies that they enforce. Other researchers have investigated optimization techniques for security automata [CF00, Thi01], certification of programs instrumented with security checks [Wal00] and the use of run-time monitoring and checking in distributed [SS98] and real-time systems [KVBA⁺99].

Overview The remainder of the paper begins with a review of Alpern and Schneider’s framework for understanding the behavior of software systems [AS87, Sch00] (Section 2) and an explanation of the EM class of security policies and security automata (Section 2.3). In Section 3 we describe our new enforcement mechanisms – insertion automata, suppression automata and edit automata. For each mechanism, we analyze the class of security policies that the mechanism is able to enforce and provide practical examples of policies that fall in that class. In Section 4 we discuss some unanswered questions and our continuing research. Section 5 concludes the paper with a taxonomy of security policies.

2 Security Policies and Enforcement Mechanisms

In this section, we explain our model of software systems and how they execute, which is based on the work of Alpern and Schneider [AS87, Sch00]. We define what it means to be a security policy and give definitions for safety, liveness and EM policies. We give a new presentation of Schneider’s security automata and their semantics that emphasizes our view of these machines as sequence transformers rather than property recognizers. Finally, we provide definitions of what it means for an automaton to *enforce* a property precisely and conservatively, and also what it means for one automaton to be a more effective enforcer than another automaton for a particular property.

2.1 Systems, Executions and Policies

We specify software systems at a high level of abstraction. A *system* $\mathcal{S} = (\mathcal{A}, \Sigma)$ is specified via a set of *program actions* $\mathcal{A}$ (also referred to as events or program operations) and a set of possible executions Σ . An *execution* σ is simply a finite sequence of actions

$a_1, a_2, \dots, a_n$. Previous authors have considered infinite executions as well as finite ones. We restrict ourselves to finite, but arbitrarily long executions to simplify our analysis. We use the metavariables σ and τ to range over finite sequences.

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 $\tau \prec \sigma$.

In this work, it will be important to distinguish between uniform systems and nonuniform systems. $(\mathcal{A}, \Sigma)$ is a *uniform* system if $\Sigma = \mathcal{A}^*$ where $\mathcal{A}^*$ is the set of all finite sequences of symbols from $\mathcal{A}$. Conversely, $(\mathcal{A}, \Sigma)$ is a *nonuniform* system if $\Sigma \subset \mathcal{A}^*$. Uniform systems arise naturally when a program is completely unconstrained; unconstrained programs may execute operations in any order. However, an effective security system will often combine static program analysis and preprocessing with run-time security monitoring. Such is the case in Java virtual machines, for example, which combine type checking with stack inspection. Program analysis and preprocessing can give rise to nonuniform systems. In this paper, we are not concerned with how nonuniform systems may be generated, be it by model checking programs, control or dataflow analysis, program instrumentation, type checking, or proof-carrying code; we care only that they exist.

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.

- *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 the policy has the following form.

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

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

Hence, a property is defined exclusively in terms of individual executions. A property may not specify a relationship between possible executions of the program. Information flow, for example, which can only be specified as a condition on a 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 implicitly assume that the empty sequence is contained in any property. For all the properties we are interested in it will always be okay not to run the program in question. From a technical perspective, this decision allows us to avoid repeatedly considering the empty sequence as a special case in future definitions of enforceable properties.

Given some set of actions $\mathcal{A}$, a predicate $\hat{P}$ over $\mathcal{A}^*$ 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 The *safety properties* are properties that specify that “nothing bad happens.” We can make this definition precise as follows. $\hat{P}$ is a safety property if and only if for all $\sigma \in \Sigma$,

$$\neg \hat{P}(\sigma) \Rightarrow \forall \sigma' \in \Sigma. (\sigma \prec \sigma' \Rightarrow \neg \hat{P}(\sigma')) \quad (\text{SAFETY})$$

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

Liveness Properties A *liveness property*, in contrast to a safety property, is a property in which nothing exceptionally bad can happen in any finite amount

of time. Any finite sequence of actions can always be extended so that it lies within the property. Formally, $\hat{P}$ is a liveness property if and only if,

$$\forall \sigma \in \Sigma. \exists \sigma' \in \Sigma. (\sigma \prec \sigma' \wedge \hat{P}(\sigma')) \quad (\text{LIVENESS})$$

Availability is a liveness property. If the program has acquired a resource, we can always extend its execution so that it releases the resource in the next step.

Other Properties Surprisingly, Alpern and Schneider [AS87] show that any property can be decomposed into the conjunction of a safety property and a liveness property. Bounded availability is a property that combines safety and liveness. For example, suppose our bounded-availability policy states that every resource that is acquired must be released and must be released at most ten steps after it is acquired. This property contains an element of safety because there is a bad thing that may occur (*e.g.*, taking 11 steps without releasing the resource). It is not purely a safety property because there are sequences that are not in the property (*e.g.*, we have taken eight steps without releasing the resource) that may be extended to sequences that are in the property (*e.g.*, we release the resource on the ninth step).

2.3 EM

Recently, Schneider [Sch00] defined a new class of security properties called EM. Informally, EM is the class of properties that can be enforced by a *monitor* that runs in parallel with a *target program*. 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 operation is allowed, the target program continues operation, and the monitor does not change the program’s behavior in any way. If the operation is not allowed, the monitor terminates execution of the program. Schneider showed that every EM property satisfies (SAFETY) and hence EM is a subset of the safety properties. In addition, Schneider considered monitors for infinite sequences and he showed that such monitors can only enforce policies that obey the following continuity property.

$$\forall \sigma \in \Sigma. \neg \hat{P}(\sigma) \Rightarrow \exists i. \neg \hat{P}(\sigma[..i]) \quad (\text{CONTINUITY})$$

Continuity states that any (infinite) execution that is not in the EM policy must have some finite prefix that is also not in the policy.

Security Automata Any EM policy can be enforced by a *security automaton* A , which is a deterministic finite or infinite state machine (Q, q_0, δ) that

is specified with respect to some system $(\mathcal{A}, \Sigma)$. Q specifies the possible automaton states and q_0 is the initial state. The partial function $\delta : \mathcal{A} \times Q \rightarrow Q$ specifies the transition function for the automaton.

Our presentation of the operational semantics of security automata deviates from the presentation given by Alpern and Schneider because we view these machines as *sequence transformers* rather than simple *sequence recognizers*. We specify the execution of a security automaton A on a sequence of program actions σ using a labeled operational semantics.

The basic single-step judgment has the form $(\sigma, q) \xrightarrow{\tau}_A (\sigma', q')$ where σ and q denote the input program action sequence and current automaton state; σ' and q' denote the action sequence and state after the automaton processes a single input symbol; and τ denotes the sequence of actions that the automaton allows to occur (either the first action in the input sequence or, in the case that this action is “bad,” no actions at all). We may also refer to the sequence τ as the observable actions or the automaton output. The input sequence σ is not considered observable to the outside world.

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

$$\begin{array}{ll}
(\sigma, q) \xrightarrow{a}_A (\sigma', q') & \text{(A-STEP)} \\
\text{if } \sigma = a; \sigma' \\
\text{and } \delta(a, q) = q' \\
(\sigma, q) \xrightarrow{\cdot}_A (\cdot, q) & \text{(A-STOP)} \\
\text{otherwise}
\end{array}$$

We extend the single-step semantics to a multi-step semantics through the following rules.

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

$$\frac{}{(\sigma, q) \Rightarrow_A (\sigma, q)} \quad \text{(A-REFLEX)}$$

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

Limitations Erlingsson and Schneider [UES99, UES00] demonstrate that security automata can enforce important access-control policies including software fault isolation and Java stack inspection. However, they cannot enforce any of our other example policies (availability, bounded availability or information flow). Schneider [Sch00] also points out that security automata cannot enforce safety properties on

systems in which the automaton cannot exert sufficient controls over the system. For example, if one of the actions in the system is the passage of time, an automaton might not be able to enforce the property because it cannot terminate an action sequence effectively — an automaton cannot stop the passage of real time.

2.4 Enforceable Properties

To be able to discuss different sorts of enforcement automata formally and to analyze how they enforce different properties, we need a formal definition of what it means for an automaton to enforce a property.

We say that an automaton A *precisely enforces* a property $\hat{P}$ on the system $(\mathcal{A}, \Sigma)$ if and only if $\forall \sigma \in \Sigma$,

1. If $\hat{P}(\sigma)$ then $\forall i. (\sigma, q_0) \xRightarrow{\sigma[1..i]}_A (\sigma[i+1..], q')$ and,
2. If $(\sigma, q_0) \xRightarrow{\sigma'}_A (\cdot, q')$ then $\hat{P}(\sigma')$

Informally, if the sequence belongs to the property $\hat{P}$ then the automaton should not modify it. In this case, we say the automaton *accepts* the sequence. If the input sequence is not in the property, then the automaton may (and in fact must) edit the sequence so that the output sequence satisfies the property.

Some properties are extremely difficult to enforce precisely, so, in practice, we often enforce a stronger property that implies the weaker property in which we are interested. For example, information flow is impossible to enforce precisely using run-time monitoring as it is not even a proper property. Instead of enforcing information flow, an automaton might enforce a simpler policy such as access control. Assuming access control implies the proper information-flow policy, we say that this automaton *conservatively enforces* the information-flow policy. Formally, an automaton conservatively enforces a property $\hat{P}$ if condition 2 from above holds. Condition 1 need not hold for an automaton to conservatively enforce a property. In other words, an automaton that conservatively enforces a property may occasionally edit an action sequence that actually obeys the policy, even though such editing is unnecessary (and potentially disruptive to the benign program’s execution). Of course, any such edits should result in an action sequence that continues to obey the policy. Henceforth, when we use the term *enforces* without qualification (precisely, conservatively) we mean *enforces precisely*.

We say that automaton A_1 enforces a property $\hat{P}$ *more precisely* or *more effectively* than another automaton A_2 when either

1. A_1 accepts more sequences than A_2 , or

2. The two automata accept the same sequences, but the average edit distance¹ between inputs and outputs for A_1 is less than that for A_2 .

3 Beyond EM

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. In this section, we describe insertion automata, suppression automata and their conjunction, edit automata. In each case, we characterize the properties they can enforce precisely.

3.1 Insertion Automata

An *insertion automaton* I is a finite or infinite state machine (Q, q_0, δ, γ) that is defined with respect to some system of executions $\mathcal{S} = (\mathcal{A}, \Sigma)$. Q is the set of all possible machine states and q_0 is a distinguished starting state for the machine. The partial function $\delta : \mathcal{A} \times Q \rightarrow Q$ specifies the transition function as before. The new element is a partial function γ that specifies the insertion of a number of actions into the program's action sequence. We call this the *insertion function* and it has type $\mathcal{A} \times Q \rightarrow \vec{\mathcal{A}} \times Q$. In order to maintain the determinacy of the automaton, we require that the domain of the insertion function is 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.

$$\begin{aligned}
 &(\sigma, q) \xrightarrow{a}_I (\sigma', q') && \text{(I-STEP)} \\
 &\text{if } \sigma = a; \sigma' \\
 &\text{and } \delta(a, q) = q' \\
 &(\sigma, q) \xrightarrow{\tau}_I (\sigma, q') && \text{(I-INS)} \\
 &\text{if } \sigma = a; \sigma' \\
 &\text{and } \gamma(a, q) = \tau, q' \\
 &(\sigma, q) \xrightarrow{\cdot}_I (\cdot, q) && \text{(I-STOP)} \\
 &\text{otherwise}
 \end{aligned}$$

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

¹The *edit distance* between two sequences is the minimum number of insertions, deletions or substitutions that must be applied to either of the sequences to make them equal [Gus97].

Enforceable Properties We will examine the power of insertion automata both on uniform systems and on nonuniform systems.

Theorem 1 (Uniform I-Enforcement) *If $\mathcal{S}$ is a uniform system and insertion automaton I precisely enforces $\hat{P}$ on $\mathcal{S}$ then $\hat{P}$ obeys (SAFETY).*

Proof: Proofs of the theorems in this work are contained in our Technical Report [BLW02]; we omit them here due to space constraints. ■

If we consider nonuniform systems then the insertion automaton can enforce non-safety properties. For example, consider a carefully chosen nonuniform system $\mathcal{S}'$, where the last action of every sequence is the special **stop** symbol, and **stop** appears nowhere else in $\mathcal{S}'$. By the definition of safety, we would like to enforce a property $\hat{P}$ such that $\neg \hat{P}(\tau)$ but $\hat{P}(\tau; \sigma)$. Consider processing the final symbol of τ . Assuming that the sequence τ does not end in **stop** (and that $\neg \hat{P}(\tau; \text{stop})$), our insertion automaton has a safe course of action. After seeing τ , our automaton waits for the next symbol (which must exist, since we asserted the last symbol of τ is not **stop**). If the next symbol is **stop**, it inserts σ and stops, thereby enforcing the policy. On the other hand, if the program itself continues to produce σ , the automaton need do nothing.

It is normally a simple matter to instrument programs so that they conform to the nonuniform system discussed above. The instrumentation process would insert a **stop** event before the program exits. Moreover, to avoid the scenario in which a non-terminating program sits in a tight loop and never commits any further security-relevant actions, we could ensure that after some time period, the automaton receives a timeout signal which also acts as a **stop** event.

Bounded-availability properties, which are not EM properties, have the same form as the policy considered above, and as a result, an insertion automaton can enforce many bounded-availability properties on non-uniform systems. In general, the automaton monitors the program as it acquires and releases resources. Upon detecting the bound, the automaton inserts actions that release the resources in question. It also releases the resources in question if it detects termination via a **stop** event or timeout.

We characterize the properties that can be enforced by an insertion automaton as follows.

Theorem 2 (Nonuniform I-Enforcement) *A property $\hat{P}$ on the system $\mathcal{S} = (\mathcal{A}, \Sigma)$ can be enforced by some insertion automaton if and only if there exists a function γ_p such that for all executions $\sigma \in \mathcal{A}^*$, if $\neg \hat{P}(\sigma)$ then*

1. $\forall \sigma' \in \Sigma. \sigma \prec \sigma' \Rightarrow \neg \hat{P}(\sigma'), \text{ or}$
2. $\sigma \notin \Sigma \text{ and } \hat{P}(\sigma; \gamma_p(\sigma))$

Limitations Like the security automaton, the insertion automaton is limited by the fact that it may not be able to exert sufficient controls over a system. More precisely, it may not be possible for the automaton to synthesize certain events and inject them into the action stream. For example, an automaton may not have access to a principal's private key. As a result, the automaton may have difficulty enforcing a fair exchange policy that requires two computational agents to exchange cryptographically signed documents. Upon receiving a signed document from one agent, the insertion automaton may not be able to force the other agent to sign the second document and it cannot forge the private key to perform the necessary cryptographic operations itself.

3.2 Suppression Automata

A *suppression automaton* S is a state machine (Q, q_0, δ, ω) that is defined with respect to some system of executions $\mathcal{S} = (\mathcal{A}, \Sigma)$. 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 $\omega : \mathcal{A} \times Q \rightarrow \{-, +\}$ has the same domain as δ and indicates whether or not the action in question is to be suppressed $(-)$ or emitted $(+)$.

$$(\sigma, q) \xrightarrow{a}_S (\sigma', q') \quad (\text{S-STEP A})$$

$$\begin{aligned} &\text{if } \sigma = a; \sigma' \\ &\text{and } \delta(a, q) = q' \\ &\text{and } \omega(a, q) = + \end{aligned}$$

$$(\sigma, q) \xrightarrow{\cdot}_S (\sigma', q') \quad (\text{S-STEPS})$$

$$\begin{aligned} &\text{if } \sigma = a; \sigma' \\ &\text{and } \delta(a, q) = q' \\ &\text{and } \omega(a, q) = - \end{aligned}$$

$$(\sigma, q) \xrightarrow{\cdot}_S (\cdot, q) \quad (\text{S-STOP})$$

otherwise

We extend the single-step relation to a multi-step relation using the reflexivity and transitivity rules from above.

Enforceable Properties In a uniform system, suppression automata can only enforce safety properties.

Theorem 3 (Uniform S-Enforcement) *If $\mathcal{S}$ is a uniform system and suppression automaton S precisely enforces $\hat{P}$ on $\mathcal{S}$ then $\hat{P}$ obeys (SAFETY).*

In a nonuniform system, suppression automata can once again enforce non-EM properties. For example, consider the following system $\mathcal{S}$.

$$\begin{aligned} \mathcal{A} &= \{\text{aq, use, rel}\} \\ \Sigma &= \{\text{aq; rel,} \\ &\quad \text{aq; use; rel,} \\ &\quad \text{aq; use; use; rel}\} \end{aligned}$$

The symbols **aq**, **use**, **rel** denote acquisition, use and release of a resource. The set of executions includes zero, one, or two uses of the resource. Such a scenario might arise were we to publish a policy that programs can use the resource at most two times. After publishing such a policy, we might find a bug in our implementation that makes it impossible for us to handle the load we were predicting. Naturally we would want to tighten the security policy as soon as possible, but we might not be able to change the policy we have published. Fortunately, we can use a suppression automaton to suppress extra uses and dynamically change the policy from a two-use policy to a one-use policy. Notice that an ordinary security automaton is not sufficient to make this change because it can only terminate execution.² After terminating a two-use application, it would be unable to insert the release necessary to satisfy the policy.

We can also compare the power of suppression automata with insertion automata. A suppression automaton cannot enforce the bounded-availability policy described in the previous section because it cannot insert release events that are necessary if the program halts prematurely. That is, although the suppression automaton could suppress all non-release actions upon reaching the bound (waiting for the release action to appear), the program may halt without releasing, leaving the resource unreleased. Note also that the suppression automaton cannot simply suppress resource acquisitions and uses because this would modify sequences that actually do satisfy the policy, contrary to the definition of precise enforcement. Hence, insertion automata can enforce some properties that suppression automata cannot.

²Premature termination of these executions takes us outside the system $\mathcal{S}$ since the **rel** symbol would be missing from the end of the sequence. To model the operation of a security automaton in such a situation we would need to separate the set of possible input sequences from the set of possible output sequences. For the sake of simplicity, we have not done so in this paper.

For any suppression automaton, we can construct an insertion automaton that enforces the same property. The construction proceeds as follows. While the suppression automaton acts as a simple security automaton, the insertion automaton can clearly simulate it. When the suppression automaton decides to suppress an action a , it does so because there exists some extension σ of the input already processed (τ) such that $\hat{P}(\tau; \sigma)$ but $\neg \hat{P}(\tau; a; \sigma)$. Hence, when the suppression automaton suppresses a (giving up on precisely enforcing any sequence with $\sigma; a$ as a prefix), the insertion automaton merely inserts σ and terminates (also giving up on precise enforcement of sequences with $\sigma; a$ as a prefix). Of course, in practice, if σ is uncomputable or only intractably computable from τ , suppression automata are useful.

There are also many scenarios in which suppression automata are *more precise* enforcers than insertion automata. In particular, in situations such as the one described above in which we publish one policy but later need to restrict it due to changing system requirements or policy bugs, we can use suppression automata to suppress resource requests that are no longer allowed. Each suppression results in a new program action stream with an edit distance increased by 1, whereas the insertion automaton may produce an output with an arbitrary edit distance from the input.

Before we can characterize the properties that can be enforced by a suppression automaton, we must generalize our suppression functions so they act over sequences of symbols. Given a set of actions $\mathcal{A}$, a computable function $\omega^* : \mathcal{A}^* \rightarrow \mathcal{A}^*$ is a *suppression function* if it satisfies the following conditions.

1. $\omega^*(\cdot) = \cdot$
2. $\omega^*(\sigma; a) = \omega^*(\sigma); a$, or $\omega^*(\sigma; a) = \omega^*(\sigma)$

A suppression automaton can enforce the following properties.

Theorem 4 (Nonuniform S-Enforcement) *A property $\hat{P}$ on the system $\mathcal{S} = (\mathcal{A}, \Sigma)$ is enforceable by a suppression automaton if and only if there exists a suppression function ω^* such that for all sequences $\sigma \in \mathcal{A}^*$,*

- if $\hat{P}(\sigma)$ then $\omega^*(\sigma) = \sigma$, and
- if $\neg \hat{P}(\sigma)$ then

1. $\forall \sigma' \in \Sigma.\sigma \prec \sigma' \Rightarrow \neg \hat{P}(\sigma')$, or
2. $\sigma \notin \Sigma$ and $\forall \sigma' \in \Sigma.\sigma \prec \sigma' \Rightarrow \hat{P}(\omega^*(\sigma'))$

Limitations Similarly to its relatives, a suppression automaton is limited by the fact that some events may not be suppressible. For example, the program may have a direct connection to some output device and the automaton may be unable to interpose itself between the device and the program. It might also be the case that the program is unable to continue proper execution if an action is suppressed. For instance, the action in question might be an input operation.

3.3 Edit Automata

We form an *edit automaton* E by combining the insertion automaton with the suppression automaton. Our machine is now described by a 5-tuple with the form $(Q, q_0, \delta, \gamma, \omega)$. The operational semantics are derived from the composition of the operational rules from the two previous automata.

$$(\sigma, q) \xrightarrow{a}_E (\sigma', q') \quad (\text{E-STEP A})$$

$$\begin{aligned} &\text{if } \sigma = a; \sigma' \\ &\text{and } \delta(a, q) = q' \\ &\text{and } \omega(a, q) = + \end{aligned}$$

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

$$\begin{aligned} &\text{if } \sigma = a; \sigma' \\ &\text{and } \delta(a, q) = q' \\ &\text{and } \omega(a, q) = - \end{aligned}$$

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

$$\begin{aligned} &\text{if } \sigma = a; \sigma' \\ &\text{and } \gamma(a, q) = \tau, q' \end{aligned}$$

$$(\sigma, q) \xrightarrow{\cdot}_E (\cdot, q) \quad (\text{E-STOP})$$

otherwise

We again extend this single-step semantics to a multi-step semantics with the rules for reflexivity and transitivity.

Enforceable Properties As with insertion and suppression automata, edit automata are only capable of enforcing safety properties in uniform systems.

Theorem 5 (Uniform E-Enforcement) *If $\mathcal{S}$ is a uniform system and edit automaton E precisely enforces $\hat{P}$ on $\mathcal{S}$ then $\hat{P}$ obeys (SAFETY).*

The following theorem provides the formal basis for the intuition given above that insertion automata are strictly more powerful than suppression automata. Because insertion automata enforce a superset of properties enforceable by suppression automata, edit automata (which are a composition of insertion and suppression automata) precisely enforce exactly those properties that are precisely enforceable by insertion automata.

Theorem 6 (Nonuniform E-Enforcement) *A property $\hat{P}$ on the system $\mathcal{S} = (\mathcal{A}, \Sigma)$ can be enforced by some edit automaton if and only if there exists a function γ_p such that for all executions $\sigma \in \mathcal{A}^*$, if $\neg \hat{P}(\sigma)$ then*

1. $\forall \sigma' \in \Sigma.\sigma \prec \sigma' \Rightarrow \neg \hat{P}(\sigma')$, or
2. $\sigma \notin \Sigma$ and $\hat{P}(\sigma; \gamma_p(\sigma))$

Although edit automata are no more powerful precise enforcers than insertion automata, we can very effectively enforce a wide variety of security policies conservatively with edit automata. We describe a particularly important application, the implementation of transactions policies, in the following section.

3.4 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 suppressing input actions from the start of the transaction. 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 outputting 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.

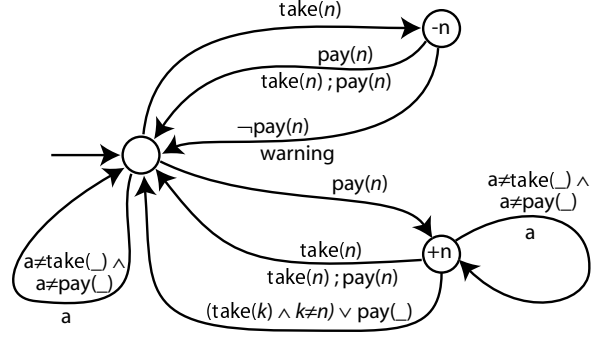


Figure 1: An edit automaton to enforce the market policy conservatively.

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 other actions that might occur in the system (such as **window-shop** or **browse**). 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. The automaton conservatively enforces 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 **pay**-**take** 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 1 displays the edit automaton that conservatively enforces our market policy. The nodes in the picture represent the automaton states and the arcs represent the transitions. When a predicate above an arc is true, the transition will be taken. The sequence below an arc represents the actions that are emitted. Hence, an arc with no symbols below it is a suppression transition. An arc with multiple symbols below it is an insertion transition.

4 Future Work

We are considering a number of directions for future research.

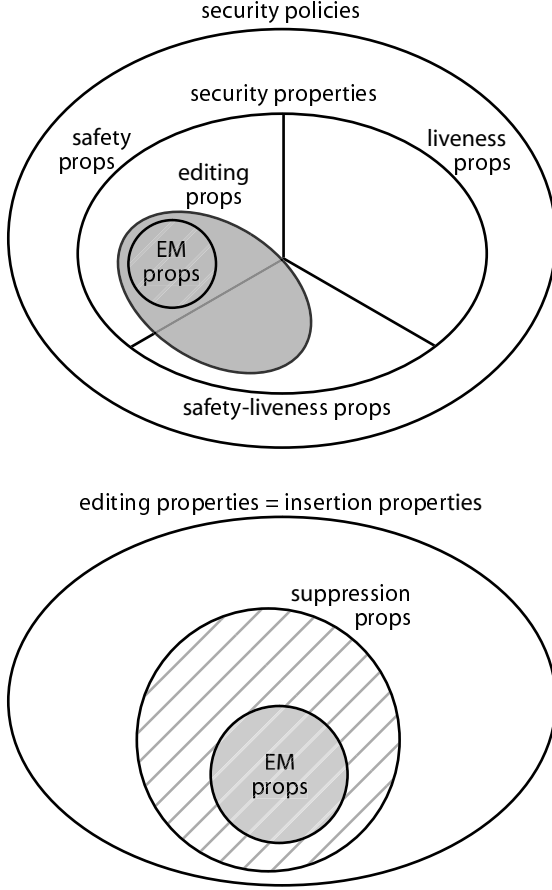


Figure 2: A taxonomy of *precisely* enforceable security policies.

Composing Schneider’s security automata is straightforward [Sch00], but this is not the case for our edit automata. Since edit automata are sequence transformers, we can easily define the composition of two automata E_1 and E_2 to be the result of running E_1 on the input sequence and then running E_2 on the output of E_1 . Such a definition, however, does not always give us the conjunction of the properties enforced by E_1 and E_2 . For example, E_2 might insert a sequence of actions that violates E_1 . When two automata operate on disjoint sets of actions, we can run one automaton after another without fear that they will interfere with one other. However, this is not generally the case. We are considering static analysis of automaton definitions to determine when they can be safely composed.

Our definitions of precise and conservative enforcement provide interesting bounds on the strictness with which properties can be enforced. Although precise enforcement of a property is most desirable because benign executions are guaranteed not to be disrupted

by edits, disallowing even provably benign modifications restricts many useful transformations (for example, the enforcement of the market policy from Section 3.4). Conservative enforcement, on the other hand, allows the most freedom in how a property is enforced because *every* property can be conservatively enforced by an automaton that simply halts on all inputs (by our assumption that $\hat{P}(\cdot)$). We are working on defining what it means to *effectively* enforce a property. This definition may place requirements on exactly what portions of input sequences must be examined, but will be less restrictive than precise enforcement and less general than conservative enforcement. Under such a definition, we hope to provide formal proof for our intuition that suppression automata effectively enforce some properties not effectively enforceable with insertion automata and vice versa, and that edit automata effectively enforce more properties than either insertion or suppression automata alone.

We found that edit automata could enforce a variety of non-EM properties on nonuniform systems, but could enforce no more properties on uniform systems than ordinary security automata. This indicates that it is essential to combine static program analysis and instrumentation with our new enforcement mechanisms if we wish to enforce the rich variety of properties described in this paper. We are very interested in exploring the synergy between these two techniques.

In practice, benign applications must be able to react to the actions of edit automata. When a program event is suppressed or inserted, the automaton must have a mechanism for signaling the program so that it may recover from the anomaly and continue its job (whenever possible). It seems likely that the automaton could signal an application with a security exception which the application can then catch, but we need experience programming in this new model.

We are curious as to whether or not we can further classify the space of security policies that can be enforced at runtime by 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 polynomial-time monitors but not linear-time monitors? What if we limit of the program’s access to random bits and therefore its ability to use strong cryptography?

We are eager to begin an implementation of a realistic policy language for Java based on the ideas described in this paper, but we have not done so yet.

5 Conclusions

In this paper we have defined two new classes of security policies that can be enforced by monitoring

programs at runtime. These new classes were discovered by considering the effect of standard editing operations on a stream of program actions. Figure 2 summarizes the relationship between the taxonomy of security policies discovered by Alpern and Schneider [AS87, Sch00] and our new editing properties.

Acknowledgment

The authors are grateful to Fred Schneider for making helpful comments and suggestions on an earlier version of this work.

References

- [AS87] Bowen Alpern and Fred Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1987.
- [BLW02] Lujio Bauer, Jarred Ligatti, and David Walker. More enforceable security policies. Technical Report TR-649-02, Princeton University, June 2002.
- [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.
- [Gus97] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.
- [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.
- [Lam85] Leslie Lamport. Logical foundation. *Lecture Notes in Computer Science*, 190:119–130, 1985.
- [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.
- [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.