Enforceable Security Policies[1]

Fred B. Schneider

Department of Computer Science
Cornell University
Ithaca, New York 14853

January 15, 1998
Revised July 24, 1999

[1]Supported in part by ARPA/RADC grant F30602-96-1-0317, AFOSR grant F49620-94-1-0198, Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Material Command, USAF, under agreement number F30602-99-1-0533, National Science Foundation Grant 9703470, and a grant from Intel Corporation. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright annotation thereon.

Abstract

A precise characterization is given for the class of security policies enforceable with mechanisms that work by monitoring system execution, and automata are introduced for specifying exactly that class of security policies. Techniques to enforce security policies specified by such automata are also discussed.

1 Introduction

A *security policy* defines execution that, for one reason or another, has been deemed unacceptable. For example, a security policy might concern

- *access control*, and restrict what operations principals can perform on objects,
- *information flow*, and restrict what principals can infer about objects from observing system behavior, or
- *availability*, and restrict principals from denying others the use of a resource.

To date, general-purpose security policies, like those above, have attracted the most attention. But application-dependent and special-purpose security policies are increasingly important [5, 8, 13, 15, 25, 27, 34]. A system to support mobile code, like Java [11], might prevent information leakage by enforcing a security policy that bars messages from being sent after files have been read. To support electronic commerce, a security policy might prohibit executions in which a customer pays for a service but the seller does not provide that service. And finally, electronic storage and retrieval of intellectual property is governed by rights-management schemes that restrict not only the use of stored materials but also the use of any derivatives [31].

The value of application-dependent and special-purpose security policies is perhaps best explained in terms of the Principle of Least Privilege [29], which holds that each principal be accorded the minimum access needed to accomplish its task. Clearly, richer notions of "minimum access" allow the Principle of Least Privilege to discriminate better between those actions that should and those that should not be allowed. Application-dependent security policies can depend on an application's state along with the semantics of that application's abstractions, so richer prescriptions for "minimum access" now become possible. In contrast, operating system abstractions—the traditional vocabulary for security policies—constitute a coarse basis for prescribing "minimum access," often forcing security policies to be approximations for what is desired.

The practicality of any security policy depends on whether that policy is enforceable and at what cost. This paper addresses those questions for the class of enforcement mechanisms that work by monitoring execution steps of a *target* and terminating execution that is about to violate the security policy being enforced. We call this class EM, for Execution Monitoring. EM includes security kernels, reference monitors, firewalls, and most other operating system and hardware-based enforcement mechanisms that have appeared in the literature. Our targets may be objects, modules, processes, subsystems, or entire systems; the execution steps monitored may range from fine-grained actions (such as memory accesses) to higher-level operations (such as method calls) to operations that change the security-configuration and thus restrict subsequent execution.

Mechanisms that use more information than becomes available only from observing the steps of a target's execution are, by definition, excluded from EM. Information provided to an EM mechanism is thus insufficient for predicting future steps the target might take, alternative possible executions, or all possible target executions. Therefore, compilers and theorem-provers, which analyze a static representation of a target to deduce information about all of its possible executions, are not EM mechanisms. The availability of information about future execution, about alternative possible executions, or about all possible target executions gives power to an enforcement mechanism—just how much power is an open question.

Also outside EM are mechanisms that modify a target before executing it. The modified target would have to be "equivalent" to the original, except for aborting executions that would violate the security policy of interest. A definition for "equivalent" is thus required to analyze this class of mechanisms.

A formal characterization of what can and cannot be accomplished using mechanisms in EM has both practical and theoretical utility. Clearly, such a characterization can inform system builders' selections of enforcement mechanisms by circumscribing the intrinsic limits of reference monitors and derivative mechanisms. From a theoretical perspective, the characterization constitutes a first step towards a taxonomy of security policies that is based on a mathematical semantics of programs. Two other classes in that taxonomy might come from relaxing EM's defining restrictions: (i) a class of enforcement mechanisms that have access to some (perhaps incomplete) representation of the target, and (ii) a class of enforcement mechanisms that modify the target before execution.

We proceed as follows. In §2, a precise characterization is given for security policies that can be enforced using mechanisms in EM. An automata-based formalism for specifying those security policies is the subject of §3. Mechanisms in EM for enforcing security policies specified by automata are described in §4. Next, §5 discusses some pragmatic issues related to specifying and enforcing security policies as well as the application of our enforcement mechanisms to safety-critical systems. The appendix contains a summary of the notation used in the paper.

2 Characterizing EM Enforcement Mechanisms

We represent target executions by finite and infinite sequences, where Ψ denotes a universe of all possible finite and infinite sequences. The manner in which executions are represented is irrelevant here. Finite and infinite sequences of atomic actions, of higher-level system steps, of program states, or of state/action pairs are all plausible alternatives. A target S defines a subset Σ_S of Ψ corresponding to the executions of S.

A characterization of EM-enforceable security policies is interesting only if the definition being used for "security policy" is broad enough so that it does not exclude things usually considered security policies.[1] Also, the definition must be independent of how EM is defined, for otherwise the characterization of EM-enforceable security policies would be a tautology, hence uninteresting. We therefore define a *security policy* to be a set of executions, specifying security policies by predicates on sets of executions. This definition is both broad[2] (giving at least as much power for defining computations disallowed by security policies as for specifying the computations possible by targets) and corresponds to the intuition that security policies rule out target executions that are deemed unacceptable. A target S *satisfies* security policy $\mathcal{P}$ if and only if $\mathcal{P}(\Sigma_S)$ equals *true*.

Given a security policy $\mathcal{P}$ and sets Σ and Π of executions, we do not require that if Σ satisfies $\mathcal{P}$ and $\Pi \subset \Sigma$ holds, then Π satisfies $\mathcal{P}$. Imposing such a requirement on security policies would disqualify interesting candidates. For instance, the requirement would preclude information flow (as defined informally in §1) from being considered a security policy—universe Ψ of all executions satisfies information flow, but a subset Π containing only those executions in which the value of a variable x in each execution is correlated with the value of y (say) clearly violates that information flow policy.

Safety Properties and EM Enforceability

By definition, enforcement mechanisms in EM work by monitoring execution of the target. Thus, any security policy $\mathcal{P}$ that can be enforced using a

[1] However, there is no harm in being liberal about what is considered a security policy.
[2] The definition clearly subsumes the noninterference-based definition of security policy in [12].

mechanism from EM must be specified by a predicate of the form

$$\mathcal{P}(\Pi): \ (\forall \sigma \in \Pi: \ \widehat{\mathcal{P}}(\sigma)) \tag{1}$$

where $\widehat{\mathcal{P}}$ is a predicate on (individual) executions. $\widehat{\mathcal{P}}$ formalizes the criteria used by the enforcement mechanism for deciding to terminate an execution that would otherwise violate the policy being enforced. In [1] and the literature on linear-time concurrent program verification, a set of executions is called a *property* if set membership is determined by each element alone and not by other members of the set. Using that terminology, we conclude from (1) that a security policy must be a property in order for that policy to have an enforcement mechanism in EM.

Not every security policy is a property. Some security policies cannot be defined using criteria that individual executions must each satisfy in isolation. For example, the information flow policy discussed above characterizes sets that are not properties (as proved in [22][3]). Whether information flows from variable x to y in a given execution depends, in part, on what values y takes in other possible executions (and whether those values are correlated with the value of x). A predicate to specify such sets of executions cannot be constructed only using predicates defined on single executions in isolation.

Not every property is EM-enforceable. Enforcement mechanisms in EM cannot base decisions on possible future execution, since that information is, by definition, not available to such a mechanism, and this further restricts what security policies can be enforced by EM mechanisms. Consider security policy $\mathcal{P}$ of (1), and suppose σ' is the prefix of some finite or infinite execution σ where $\widehat{\mathcal{P}}(\sigma) = \mathit{true}$ and $\widehat{\mathcal{P}}(\sigma') = \mathit{false}$ hold. Because execution of a target might terminate before σ' is extended into σ, an enforcement mechanism for $\mathcal{P}$ must prohibit σ' (even though supersequence σ satisfies $\widehat{\mathcal{P}}$).

We can formalize this requirement as follows. For σ a finite or infinite execution having i or more steps, and τ' a finite execution, let

$\sigma[..i]$ denote the prefix of σ involving its first i steps

$\tau'\,\sigma$ denote execution τ' followed by execution σ

and define Π^- to be the set of all finite prefixes of elements in set Π of finite and/or infinite sequences. Then, the above requirement for $\mathcal{P}$—that $\mathcal{P}$ is *prefix closed*—is:

$$(\forall \tau' \in \Psi^-: \ \neg\widehat{\mathcal{P}}(\tau') \Rightarrow (\forall \sigma \in \Psi: \ \neg\widehat{\mathcal{P}}(\tau'\sigma))) \tag{2}$$

[3]The author of [22] acknowledged James Gray III as pointing out this limitation for dealing with security in frameworks based on our property abstraction.

Finally, note that any execution rejected by an enforcement mechanism must be rejected after a finite period. This is formalized by:

$$(\forall \sigma \in \Psi : \neg \widehat{\mathcal{P}}(\sigma) \Rightarrow (\exists i : \neg \widehat{\mathcal{P}}(\sigma[..i]))) \tag{3}$$

Security policies satisfying (1), (2), and (3) are *safety properties* [17], properties stipulating that no "bad thing" happens during any execution. Formally, a property Γ is defined in [18] to be a safety property if and only if, for any finite or infinite execution σ,

$$\sigma \notin \Gamma \Rightarrow (\exists i\colon\ (\forall \tau \in \Psi\colon\ \sigma[..i]\, \tau \notin \Gamma)) \tag{4}$$

holds. This means that Γ is a safety property if and only if Γ can be characterized using a set of finite executions that are prefixes of all executions excluded from Γ. Clearly, a security policy $\mathcal{P}$ satisfying (1), (2), and (3) has such a set of finite prefixes—the set of prefixes $\tau' \in \Psi^-$ such that $\neg\widehat{\mathcal{P}}(\tau')$ holds—so $\mathcal{P}$ is satisfied by sets that are safety properties according to (4).

The above analysis of enforcement mechanisms in EM has established:

Non EM-Enforceable Security Policies: If the set of executions for a security policy $\mathcal{P}$ is not a safety property, then an enforcement mechanism from EM does not exist for $\mathcal{P}$.

Obviously, the contrapositive holds as well: EM enforcement mechanisms enforce security policies that are safety properties. But, as discussed later in §4, the converse—that all safety properties have EM enforcement mechanisms—does not hold.

One consequence of our Non EM-Enforceable Security Policies result is that ruling-out additional executions never causes an EM-enforceable policy to be violated, since ruling-out executions never invalidates a safety property. Thus, an EM enforcement mechanism for any security policy $\mathcal{P}'$ satisfying $\mathcal{P}' \Rightarrow \mathcal{P}$ also enforces security policy $\mathcal{P}$. However, a stronger policy $\mathcal{P}'$ might proscribe executions that do not violate $\mathcal{P}$, so using $\mathcal{P}'$ is not without potentially significant adverse consequences. The limit case, where $\mathcal{P}'$ specifies the empty set, illustrates this problem.

Second, our Non EM-Enforceable Security Policies result implies that EM mechanisms compose in a natural way. When multiple EM mechanisms are used in tandem, the policy enforced by the aggregate is the conjunction of the policies that are enforced by each mechanism in isolation. This is attractive, because it enables complex policies to be decomposed into conjuncts, with a separate mechanism used to enforce each of the component policies.

Revisiting the three application-independent security policies described in §1, we find:

- Access control defines safety properties. The set of proscribed partial executions contains those partial executions ending with an unacceptable operation being attempted.
- Information flow does not define sets that are properties (as argued above), so it does not define sets that are safety properties. Not being safety properties, there are no EM enforcement mechanisms for exactly this policy.[4]
- Availability, if taken to mean that no principal is forever denied use of some given resource, is not a safety property—any partial execution can be extended in a way that allows a principal to access the resource, so the defining set of proscribed partial executions that every safety property must have is absent. In [9], availability is defined to rule out all denials in excess of MWT seconds (for some predefined Maximum Waiting Time parameter MWT). This is a safety property; the defining set of partial executions contains prefixes ending in intervals that exceed MWT seconds during which a principal is denied use of a resource.

3 Security Automata

Enforcement mechanisms in EM work by terminating target execution that is described by a finite prefix σ' such that $\neg\widehat{\mathcal{P}}(\sigma')$ holds, for a predicate $\widehat{\mathcal{P}}$ defined by the policy being enforced. In addition, we established in §2 that the set of executions satisfying $\widehat{\mathcal{P}}$ must be a safety property. Those being the only constraints on $\widehat{\mathcal{P}}$, we conclude that recognizers for sets of executions that are safety properties can serve as the basis for enforcement mechanisms in EM.

A class of Büchi automata [6] that accept safety properties was introduced (although not named) in [2]. We shall here refer to these recognizers as *security automata*; they are similar to ordinary non-deterministic finite-state automata [14]. Formally, a security automaton is defined by:

[4] Mechanisms from EM purporting to prevent information flow do so by enforcing a security policy that implies, but is not equivalent to, the absence of information flow. And, there do exist security policies that both imply restrictions on information flow and define sets that are safety properties.

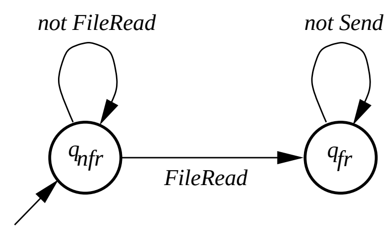

Figure 1: No *Send* after *FileRead*

- a countable set Q of *automaton states*,
- a countable set $Q_0 \subseteq Q$ of *initial automaton states*,
- a countable set I of *input symbols*, and
- a *transition function*[5], δ: $(Q \times I) \rightarrow 2^Q$.

Set I of input symbols is dictated by the security policy being enforced and the manner in which target executions are being represented; the symbols in I might correspond to system states, atomic actions, higher-level actions of the system, or state/action pairs.

To process a sequence $s_1 s_2 \ldots$ of input symbols, the *current state* Q' of the security automaton starts equal to Q_0 and the sequence is read one input symbol at a time. As each input symbol s_i is read, the security automaton changes Q' to:

$$\bigcup_{q \in Q'} \delta(q, s_i).$$

If Q' is ever the empty set, the input is rejected; otherwise the input is accepted. Notice that this acceptance criterion means that a security automaton can accept sequences that have infinite length as well as those having finite length.

Figure 1 depicts a security automaton for a security policy that prohibits execution of *Send* operations after a *FileRead* has been executed. In this diagram, the automaton states are represented by the two nodes labeled q_{nfr} (for "no file read") and q_{fr} (for "file read"). Initial states of the automaton are represented in the diagram by nodes with unlabeled incoming edges, so automaton state q_{nfr} is the only initial automaton state. Transition function δ is specified in terms of edges labeled by *transition predicates*, which are

[5]Notation 2^Q denotes the power set for set Q.

$$
\begin{array}{ll}
\textbf{state vars}: & state \;:\; \{0,1\} \;\; \textbf{initial } 0 \\[4pt]
\textbf{transitions}: & \textbf{not } FileRead \wedge state = 0 \longrightarrow \textbf{skip} \\
 & FileRead \wedge state = 0 \longrightarrow state := 1 \\
 & \textbf{not } Send \wedge state = 1 \longrightarrow \textbf{skip}
\end{array}
$$

Figure 2: Alternative specification for policy: No *Send* after $FileRead$

Boolean-valued effectively computable total functions with domain I. Let p_{ij} denote the predicate that labels the edge from node q_i to node q_j. Then, the security automaton, upon reading an input symbol s, Q' is set to

$$\{q_j \mid q_i \in Q' \ \wedge \ s \models p_{ij}\}.$$

In Figure 1, transition predicate *not FileRead* is assumed to be satisfied by input symbols (system execution steps) that are not file read operations, and transition predicate *not Send* is assumed to be satisfied by input symbols that are not message-send operations. Since no transition is defined from q_{fr} for input symbols corresponding to message-send execution steps, the security automaton in Figure 1 rejects inputs in which a *Send* follows a $FileRead$.

Diagrams like Figure 1 are impractical to draw and hard to understand if set Q of automaton states is large or transition function δ is complex. We can avoid these difficulties by encoding Q' for an automaton in multiple variables and by using *guarded commands* [4] to describe the transition function for the security automaton. Guarded command

$$B \longrightarrow S \tag{5}$$

specifies that the state transition defined by program fragment S occurs whenever predicate B is satisfied by the current input symbol and the current state of the automaton. In (5), B is called the *guard*, and it is a predicate that can refer only to the current input symbol and to the variables encoding the current state of the automaton; S is called the *command*, and it is a computation that updates (only) the variables encoding the current state of the automaton.

To illustrate this alternative notation for security automata, Figure 2 gives a specification for the same security policy as given in Figure 1. The

$$
\begin{array}{ll}
\textbf{state vars}: & A \;:\; \textbf{array}[\mathit{PRINS}, \mathit{OBJS}] \textbf{ of set of } \mathit{RIGHTS} \\[1ex]
\textbf{transitions}: & \mathit{Access}(p, obj, oper) \;\wedge\; oper \in A[p, obj] \longrightarrow \textbf{skip} \\[1ex]
& \mathit{AddRight}(p, addP, addR, obj) \;\wedge\; cntrl \in A[p, obj] \\
& \quad \longrightarrow A[addP, obj] := A[addP, obj] \cup \{addR\} \\[1ex]
& \mathit{RmvRight}(p, rmvP, rmvR, obj) \;\wedge\; cntrl \in A[p, obj] \\
& \quad \longrightarrow A[rmvP, obj] := A[rmvP, obj] - \{rmvR\}
\end{array}
$$

Figure 3: Access Control

state vars section of this specification introduces the variables that encode the current state of the security automaton. The **transitions** section gives a list of guarded commands that define the transition function. In Figure 2, *state*—a two-valued variable with initial value 0—encodes the current state of the security automaton, and each of the three guarded commands corresponds to a single edge in the diagram of Figure 1.

As a second example, Figure 3 specifies a security automaton for a simple form of access control [19]. Here, an access control matrix A encodes the current state of the security automaton—$A[p, o]$ is the set of rights that principal p has to object o. A is defined in terms of a universe of principals PRINS, a universe of objects OBJS, and a universe of access rights RIGHTS. For simplicity, no initialization is given for A.

The transitions in the security automaton of Figure 3 are defined using predicates on input symbols that correspond to a next step of execution:

$\mathit{Access}(p, obj, oper)$: Principal p invoked an access operation *oper* naming object *obj*.

$\mathit{AddRight}(p, addP, addR, obj)$: Principal p invoked an operation to add right *addR* for principal *addP* to object *obj*.

$\mathit{RmvRight}(p, rmvP, rmvR, obj)$: Principal p invoked an operation to remove right *rmvR* for principal *rmvP* for object *obj*.

The first guarded command asserts that access operations are permitted provided a principal attempting the operation has the appropriate access right for the named object. The second and third guarded commands describe a simplified policy for granting and revoking rights to principals. In

$$
\begin{array}{ll}
\textbf{state vars}: & state \;:\; \{0,1\} \;\; \textbf{initial } 0 \\[1em]
\textbf{transitions}: & \textbf{not } Pay(C) \wedge state = 0 \longrightarrow \textbf{skip} \\[1em]
& Pay(C) \wedge state = 0 \longrightarrow state := 1 \\[1em]
& Serve(C) \wedge state = 1 \longrightarrow state := 0
\end{array}
$$

Figure 4: Security automaton for fair transaction

particular, the second (third) guarded command asserts that only principals having the *cntrl* right for an object *obj* can grant (remove) rights to other principals for accessing *obj*. More-realistic policies for changing the access control matrix are easily accommodated.

Two things are worth noting about this access-control example. First, leverage results from employing a suitable representation (an access control matrix) for the current state of the automaton. Imagine how awkward it would be to try and describe changes to principal's access rights in terms of the flat set of uninterpreted automaton states. Second, the security automaton does not distinguish security-configuration changes (i.e., changing A when access rights are added and deleted) from ordinary accesses. We would argue that there is no value in making a distinction between these different kinds of operations. This view is not universally held [10].

As a final illustration, we turn to electronic commerce. We might, for example, desire that a service-provider be prevented from engaging in actions other than delivering service for which a customer has paid. This requirement is a security policy; it can be formalized in terms of the following predicates on input symbols, if input symbols represent operation executions:

$pay(C)$: customer C requests and pays for service

$serve(C)$: customer C is rendered service

The security policy of interest proscribes executions in which the service-provider executes an operation that does not satisfy $serve(C)$ after having engaged in an operation that satisfies $pay(C)$. A security automaton for this policy is defined in Figure 4.

Notice, the security automaton of Figure 4 does not stipulate that payment guarantees service—it only limits what the service-provider can do once a customer has made payment. In particular, the security policy that is specified allows a service-provider to stop executing (i.e., stop producing input symbols) rather than rendering a paid-for service. We cannot specify the stronger security policy (that service be guaranteed after payment) because that is not a safety property—there is no defining set of proscribed partial executions.

4 Using Security Automata for Enforcement

Any security automaton can serve as the basis for an enforcement mechanism in EM. The target is executed in tandem with a simulation of the security automaton.[6] In particular, initialization or creation of the target causes an instance of the security automaton simulation to be created, with the security automaton in its initial state. And, each step the target is about to take generates an input symbol, which is sent to that simulation:

(i) If the automaton can make a transition on that input symbol, then the target is allowed to perform that step and the automaton state is changed according to its transition function.

(ii) If the automaton cannot make a transition on that input symbol, then the target is terminated (for having attempted to violate the security policy).

Implicit in this approach are some assumptions.

Bounded Memory. The memory that can be devoted to simulating a security automaton will, of necessity, be finite—real computers have finite memories. Recall from §3 that our security automata can have an infinite (countable) number of automaton states.

Infinite sets of automaton states are necessary for recognizing certain safety properties, because whether a given prefix should be rejected might depend on all of the input symbols in that prefix. The ever-larger prefixes produced as execution proceeds thus require ever-larger sets of states to encode needed information about the past. For example, a safety property stipulating that, at each step of execution, the value of some target variable x equals the sum of its values in preceding states requires (to store the sum of the past values of x) a state variable that grows without bound.

[6] A similar approach—developed independently—for integrating software components whose behaviors need to be reconciled is outlined in [21].

Security policies of concern in real systems do not seem to require large amounts of storage and, in fact, are today enforced using mechanisms that use only modest amounts of storage; a security automaton to specify such a policy would also require only a modest-sized set of automaton states. We see no reason to expect application-specific or special-purpose security policies to be different. So, restricting the **state vars** for a security automaton to a finite amount of storage is not, in practice, a limitation.

Target Control. Implicit in (ii) is the assumption that the target can be terminated by the enforcement mechanism. Specifically, we assume that the enforcement mechanism has sufficient control over the target to stop further automaton input symbols from being produced. This control requirement is subtle and makes certain security policies—even though they characterize sets that are safety properties—unenforceable using mechanisms from EM.

For example, recall from §2 the definition of availability in [9]:

Real-Time Availability: One principal cannot be denied use of a resource for more than MWT seconds.

Sets satisfying Real-Time Availability are safety properties—the "bad thing" is an interval of execution spanning more than MWT seconds during which some principal is denied the resource. The input symbols of a security automaton for Real-Time Availability will therefore encode time, and a new input symbol is produced whenever time increases.

While individual clocks might be stopped, the passage of time cannot be stopped. So the target cannot be stopped from producing input symbols. Real-Time Availability simply cannot be enforced by running an automaton simulation in tandem with a target, because targets cannot provide the necessary controls to the enforcement mechanism. And since the other mechanisms in EM are no more powerful, we conclude that Real-Time Availability cannot be enforced using any mechanism in EM. Change the specification from "MWT seconds" to "MWT execution steps" and the target can be prevented from violating the policy by stopping execution, resulting in an EM-enforceable security policy.

Enforcement Mechanism Integrity. A target that corrupts a security automaton simulation can subvert any enforcement mechanism based on that simulation: Input to the enforcement mechanism must correspond to target execution; state transitions must follow the automaton's transition function. Ensuring that input to the enforcement mechanism is both correct and complete is a question of target instrumentation and monitoring.

The "complete mediation" requirement associated with reference monitors is one way to discharge this assumption. Ensuring that the target does not interfere with automaton transitions is a matter of isolation—the enforcement mechanism must be isolated from the target. Isolation of our enforcement mechanism is accomplished if, for example, the **state vars** and **transitions** for the security automaton are not writable by the target.

Pragmatics

Two mechanisms are involved in the above security-automaton based implementation of an enforcement mechanism.

Automaton Input Read: A mechanism to determine that an input symbol has been produced by the target and then to forward that symbol to the security automaton simulation.

Automaton Transition: A mechanism to determine whether the security automaton can make a transition on a given input and then to perform that transition.

How these are implemented determines the cost of the enforcement mechanism. For example, when the automaton's input symbols are the set of target states and its transition predicates are arbitrary state predicates, a new input symbol is produced each time any component of the target's state changes. Since the program counter is a state component and it changes each time a machine-language instruction is executed or an interrupt occurs, the enforcement mechanism must be involved in executing each target instruction, and that could be quite costly.

For security policies where the target's production of automaton input symbols coincides with occurrences of hardware traps, an automata-based enforcement mechanism can be supported quite cheaply by incorporating it into the trap-handler. One example is implementing an enforcement mechanism for access control policies on operating system objects, such as files. Here, the target is a file and the production of input symbols coincides with invocations of system operations (i.e., file access operations). The production of input symbols now coincides with occurrences of system-call traps.

A second example where hardware traps can be exploited arises in implementing memory protection. Memory protection implements access control with read, write, and execute operations and an access control matrix that tells which processes can access each region of memory. The security automaton of Figure 3 specifies this security policy. Notice that this security automaton would expect an input symbol for each memory reference, though. But most of these input symbols cause no change to the security automaton's state. Input symbols that do not cause automaton state transitions need not be forwarded to the automaton, and that justifies the following optimization of Automaton Input Read:

Automaton Input Read Optimization: Input symbols are not forwarded to the security automaton if the state of the automaton just after the transition would be the same as it was before the transition.

Given this optimization, the production of automaton input symbols for memory protection can be made to coincide with occurrences of traps. The target's memory-protection hardware—base/bounds registers or page and segment tables—is initialized so that a trap occurs when an input symbol should be forwarded to the memory protection automaton. Memory references that do not cause traps never cause a state transition or undefined transition by the automaton. Note, however, if this optimization is used, then a target can subvert the enforcement mechanism by corrupting the filter that selects whether to forward an input symbol to the security automaton.

Finally, inexpensive implementation of our automata-based enforcement mechanisms is also possible when programs are executed by a software-implemented virtual machine. The virtual machine instruction-processing cycle is augmented so that it produces input symbols and makes automaton transitions, according to either an internal or an externally specified security automaton. For example, the Java virtual machine [20] could easily be augmented to implement the Automaton Input Read and Automaton Transition mechanisms for input symbols that correspond to method invocations.

Beyond EM Enforcement Mechanisms

Response to Violations. Termination of a target that is about to violate a security policy might seem draconian. Yet, by definition, this is how an EM mechanism responds to an attempted violation. Why not simply notify the target that an erroneous execution step has been attempted? The target could then substitute another step and its execution might then continue.

In terms of our security automata framework, notifying a target is equivalent to having the security automaton extend that target's execution (rather than truncating that execution). And some—but not all—security policies do allow input prefixes to be extended in this manner. A security policy that does not enjoy this attribute is the variant of Real-Time Availability given in §2 where MWT bounds the number of execution steps (not seconds) that elapse before an action is taken. Various other safety properties also do not allow execution prefixes to be extended, although their practical significance as security policies is an open question.

EM was defined to truncate execution for generality. Expanding EM to include enforcement mechanisms that handle violations by notifying the target or by truncating its execution would not change the set of security policies that are EM enforceable. Modifying EM to require enforcement mechanisms that handle violations by necessarily notifying the target would shrink the set of security policies that are EM enforceable, and with no apparent gain.

Program Modification. The overhead of enforcement can be reduced by merging the enforcement mechanism into the target. One such scheme is *software-based fault isolation* (SFI), also known as "sandboxing" [33, 30]. SFI implements memory protection, as specified by an automaton like that of Figure 3, but does so without hardware assistance.[7] Instead, a program is edited before it is executed, and only such edited programs are executed by the target. (Usually, it is the object code that is edited.) The edits insert instructions to check and/or modify the values of operands, so that illegal memory references are never attempted.

SFI is not in EM because SFI involves modifying the target, and such modifications are not permitted of enforcement mechanisms in EM. But viewed in our framework, the inserted instructions for SFI can be seen to implement Automaton Input Read by copying code for Automaton Transition in-line before each target instruction that produces an input symbol. Generalizing, nothing prevents the SFI approach from being used with arbitrary security automata, thereby enforcing any EM-enforceable security policy. Trust must be placed in the tools used to modify the target, however.

Our SASI (<u>S</u>ecurity <u>A</u>utomata <u>S</u>FI <u>I</u>mplementation) prototypes for Intel's x86 object code and SUN's JVM (Java Virtual Machine) explored the use of an SFI-like approach for EM-enforceable policies [7]. Each of our prototypes merges the simulation of a security automaton into the object code for the program that is the target. New variables—accessible only to the code added for SASI—represent the current state of a security automaton, and new code—that cannot be circumvented—simulates automaton state transitions. The new code also causes the target system to halt whenever

[7]Specifically, the security policy enforced by SFI would involve only the first guarded command of the three in Figure 3, and transition predicate $Access(p, obj, oper)$ would check that the memory address being read, written, or branched to is a legal one for the program.

the automaton rejects its input (because the current automaton state does not allow a transition for the next target instruction). Analysis of a target allows simplification of code for simulating a security automaton. Each inserted copy of the automaton simulation is a candidate for simplification based on the context in which that code appears. By using partial evaluation [16] on the guards as well as by using the automaton structure, irrelevant tests and updates to the security automaton state can be removed.

Program Analysis. There is no need for any run-time enforcement mechanism if the target can be analyzed and proved not to violate the security policy of interest. This approach has been employed for a security policy like what SFI was originally intended to address in *proof carrying code* (PCC) [24]. With PCC, a proof is supplied along with a program, and this proof comes in a form that can be checked mechanically before running that program. The security policy will not be violated if, before the program is executed, the accompanying proof is checked and found to be correct. The original formulation of PCC required that proofs be constructed by hand. This restriction can be relaxed. For certain security policies, a compiler can automatically produce PCC from programs written in high-level, type-safe programming languages[23, 26].

To extend PCC for security policies that are specified by arbitrary security automata, a method is needed to extract proof obligations for establishing that a program satisfies the property given by such an automaton. Such a method does exist—it is described in [3].

5 Discussion

The utility of a formalism partly depends on the ease with which objects of the formalism can be read and written. Users of the formalism must be able to translate informal requirements into objects of the formalism. With security automata, establishing the correspondence between transition predicates and informal requirements on system behavior is crucial and can require a detailed understanding of the target. The automaton of Figure 1, for example, only captures the informal requirement that messages are not sent after a file is read if it is impossible to send a message unless transition predicate $Send$ is *true* and it is impossible to read a file unless transition predicate $FileRead$ is *true*. There might be many ways to send messages—some obvious and others buried deep within the bowels of the target. All must be identified and included in the definition of $Send$; a similar obligation accompanies transition predicate $FileRead$.

The general problem of establishing the correspondence between informal requirements and some purported formalization of those requirements is not new to software engineers. The usual solution is to analyze the formalization, being alert to inconsistencies between the results of the analysis and the informal requirements. We might use a formal logic to derive consequences from the formalization; we might use partial evaluation to analyze what the formalization implies about one or another scenario, a form of testing; or, we might (manually or automatically) transform the formalization into a prototype and observe its behavior in various scenarios.

Success with proving, testing, or prototyping as a way to gain confidence in a formalization depends upon two things. The first is to decide what aspects of a formalization to check, and this is largely independent of the formalism. But the second, having the means to do those checks, not only depends on the formalism but largely determines the usability of that formalism. To do proving, we require a logic whose language includes the formalism; to do testing, we require a means of evaluating a formalization in one or another scenario; and to do prototyping, we must have some way to transform a formalization into a computational form.

As it happens, a rich set of analytical tools does exist for security automata, because security automata are a class of Büchi automata [6] which are widely used in computer-aided program verification tools. Existing formal methods based either on model checking or on theorem proving can be employed to analyze a security policy that has been specified as a security automaton. And, testing or prototyping a security policy that is specified by a security automaton is just a matter of running the automaton.

Guidelines for Structuring Security Automata

Real system security policies are best given as collections of simpler policies, a single large monolithic policy being difficult to comprehend. The system's security policy is then the result of composing the simpler policies in the collection by taking their conjunction. To employ such a separation of concerns when security policies are specified by security automata, we must be able to compose security automata in an analogous fashion. Given a collection of security automata, we must be able to construct a single *conjunction security automaton* for the conjunction of the security policies specified by the automata in the collection. That construction is not difficult: An execution is rejected by the conjunction security automaton if and only if it is rejected by any automaton in the collection.

Beyond comprehensibility, there are other advantages to specifying system security policies as collections of security automata. First, having a collection allows different enforcement mechanisms to be used for the different automata (hence the different security policies) in the collection. Second, security policies specified by distinct automata can be enforced by distinct system components, something that is attractive when all of a given security automaton's input symbols correspond to events at a single system component. Benefits that accrue from having the source of all of an automaton's input symbols be a single component include:

- Enforcement of a component's security policy involves trusting only that component.
- The overhead of an enforcement mechanism is lower because communication between components can be reduced.

For example, the security policy for a distributed system might be specified by giving a separate security automaton for each system host. Then, each host would itself implement Automaton Input Read and Automaton Transitions mechanisms for only the security automata concerning that host.

Application to Safety-Critical Systems

The idea that security kernels might have application in safety-critical systems is eloquently justified in [28] and continues to interest researchers [32]. Safety-critical systems are, for the most part, concerned with enforcing properties that are safety properties (in the sense of [18]), so it is natural to expect an enforcement mechanism for safety properties to have application in this class of systems. And, we see no impediments to using security automata or our security-automata based enforcement mechanisms for enforcing safety properties in safety-critical systems.

The justification given in [28] for using security kernels in safety-critical systems involves a characterization of what types of properties can be enforced by a security kernel. As do we in this paper, [28] concludes that safety properties but not liveness properties[8]are enforceable. However, the arguments given in [28] are informal and are coupled to the semantics of kernel-supported operations. The essential attributes of enforceability, which we isolate and formalize by equations (1), (2), and (3), are neither identified nor shown to imply that only safety properties can be enforced.

[8]A *liveness property* is a property that stipulates some "good thing" happens during any execution. See [2] for a formal definition.

In addition, because [28] concerns kernelized systems, the notion of property there is restricted to being sequences of kernel-provided functions. By allowing security automata to have arbitrary sets of input symbols, our results can be seen as generalizing those of [28]. And the generalization is a useful one, because it applies to enforcement mechanisms that are not part of a kernel. Thus, we can now extend the central thesis of [28], that kernelized systems have application beyond implementing security policies, to justify the use of enforcement mechanisms from EM when building safety-critical systems.

Acknowledgments

I am grateful to Robbert van Renesse, Greg Morrisett, Úlfar Erlingsson, Yaron Minsky, and Lidong Zhou for helpful feedback on the use and implementation of security automata and for comments on previous drafts of this paper. Helpful comments on earlier drafts of this paper were also provided by Earl Boebert, Dave Evans, Li Gong, Robert Grimm, Keith Marzullo, Andrew Myers, John Rushby, and Chris Small. John McLean served as a valuable sounding board for these ideas as I developed them. Feedback from Martin Abadi helped to sharpen the formalism. Reviewer C wrote a lengthy review and suggested citations [9] and [10]. And, the University of Tromso was a hospitable setting and a compelling excuse for performing some of the work reported herein.

References

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

[2] Alpern, B. and F.B. Schneider. Recognizing safety and liveness. *Distributed Computing 2* (1987), 117-126.

[3] Alpern, B. and F.B. Schneider. Verifying temporal properties without using temporal logic. *ACM Transactions on Programming Languages and Systems 11*, 1 (January 1989), 147–167.

[4] Dijkstra, E.W. Guarded commands, nondeterminacy, and formal derivation of programs. *Communications of the ACM 18*, 8 (Aug. 1975), 453–457.

[5] Edjlali, G., A. Acharya, and V. Chaudhary. History-based access control for mobile code. *Proceedings 5th Conference on Computer & Communications Security* (San Francisco, Calif., November 1998), ACM SIGSAC, 38–48.

[6] Eilenberg, S. *Automata, Languages, and Machines*. Vol. A, Academic Press, 1974, New York.

[7] Erlingsson, Ú. and F.B. Schneider. SASI Enforcement of Security Policies: A Retrospective. To appear, *Proceedings New Security Paradigms Workshop 1999*.

[8] Evans, D. and A. Twyman. Policy-directed code safety. *Proceedings 1999 IEEE Computer Society Symposium on Research in Security and Privacy* (Oakland, Calif., May 1999), IEEE Computer Society, Calif., 32–45.

[9] Gligor, V. A note on denial-of-service in operating systems. *IEEE Transactions on Software Engineering SE-10*, 3 (May 1984), 320–324.

[10] Gligor, V.D., S.I. Gavrila, and D. Ferraiolo. On the formal definition of separation-of-duty policies and their composition. *Proceedings 1998 IEEE Computer Society Symposium on Research in Security and Privacy* (Oakland, Calif., May 1998), IEEE Computer Society, Calif., 172–183.

[11] Gong, L. Java security: Present and near future. *IEEE Micro 17*, 3 (May/June 1997), 14–19.

[12] Goguen, J.A. and J. Meseguer. Security policies and security models. *Proceedings 1992 IEEE Computer Society Symposium on Research in Security and Privacy* (Oakland, Calif., May 1982), IEEE Computer Society, Calif., 11–20.

[13] Grimm, R. and B.N. Bershad. Providing policy-neutral and transparent access control in extensible systems. In *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, Lecture Notes in Computer Science, Vol 1603. J. Vitek, C.D. Jensen, editors. Springer-Verlag, 1999, New York, 317–338.

[14] Hopcroft, J. and J. Ullman. *Formal Languages and Their Relation to Automata*. Addison Wesley Publishing Company, Reading, Mass., 1969.

[15] Jajodia, S., P. Samarati, and V.S. Subrahmanian. A logical language for expressing authorizations. *Proceedings 1997 IEEE Computer Society Symposium on Research in Security and Privacy* (Oakland, Calif., May 1997), IEEE Computer Society, Calif., 31–42.

[16] Jones, N.D., C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, New Jersey, 1993.

[17] Lamport, L. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering SE-3*, 2 (March 1977), 125–143.

[18] Lamport, L. Logical Foundation. In *Distributed Systems-Methods and Tools for Specification*, Lecture Notes in Computer Science, Vol 190. M. Paul and H.J. Siegert, editors. Springer-Verlag, 1985, New York, 119–30.

[19] Lampson, B. Protection. *Proceedings 5th Symposium on Information Sciences and Systems* (Princeton, New Jersey, March 1971), 437–443. Reprinted in *Operating System Review 8*, 1 (Jan. 1974), 18–24.

[20] Lindholm, T. and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Menlo Park, California, 1997.

[21] Marchukov, M. and K. Sullivan. Reconciling behavioral mismatch through component restriction. Technical Report CS 99-22, Department of Computer Science, University of Virginia, July 1999.

[22] McLean, J. A general theory of composition for trace sets closed under selective interleaving functions. *Proceedings 1994 IEEE Computer Society Symposium on Research in Security and Privacy* (Oakland, Calif., May 1994), IEEE Computer Society, Calif., 79–93.

[23] Morrisett, G., D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *Proceedings 25th Annual Symposium on Principles of Programming Languages* (San Diego, Calif., Jan. 1998), ACM, New York, 85–97.

[24] Necula, G. Proof-carrying code. *Proceedings 24th Annual Symposium on Principles of Programming Languages* (Paris, France, Jan. 1997), ACM, New York, 106–119.

[25] Null, L.M. and J. Wong. The DIAMOND security policy for object-oriented databases. *Proceedings of the 1992 ACM Computer Science*

20th Annual Conference on Communications (Kansas City, Missouri, March 1992), ACM, New York, 49–56.

[26] Necula, G.C. and P. Lee. The design and implementation of a certifying compiler. *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation* (Montreal, Canada, June 1998), ACM, New York, 333–344.

[27] Pandey, R. and B. Hashii. Providing fine grained access control for mobile programs through binary editing. Technical Report TR98 08, University of California, Davis, August 1998.

[28] Rushby, J. Kernels for safety? In *Safe and Secure Computing Systems*, T. Anderson, editor. Blackwell Scientific Publications, 1989, 210–220.

[29] Saltzer J.H. and M.D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE 63*, 9 (Sept. 1975), 1278–1308.

[30] Small, C. MiSFIT: A tool for constructing safe extensible C++ systems. *Proceedings of the Third USENIX Conference on Object-Oriented Technologies* (Portland, Oregon, June 1997), USENIX.

[31] Stefik, M. Letting Loose the Light: Igniting Commerce in Electronic Publication. In *Internet Dreams*, M. Stefik, editor, MIT Press, 1996.

[32] Wika, K. G. and J. C. Knight. On the enforcement of software safety policies. *Proceedings 10th Annual IEEE Conference on Computer Assurance* (Gaithersburg, Maryland, June 1995), IEEE Computer Society, Calif.

[33] Wahbe, R., S. Lucco, T.E. Anderson, and S. L. Graham. Efficient Software-Based Fault Isolation. *Proceeding 14th ACM Symposium on Operating Systems Principles* (Asheville, North Carolina, Dec. 1993), ACM, New York, 202–216.

[34] Woo, T.Y.C. and S.S. Lam. Authorization in distributed systems: A formal approach. *Proceedings 1992 IEEE Computer Society Symposium on Research in Security and Privacy* (Oakland, Calif., May 1992), IEEE Computer Society, Calif., 33–50.

Appendix: Summary of Notation

Ψ: The set of all finite and infinite sequences.

S: A target.

Σ_S: The set of executions possible by target S.

$\mathcal{P}$: A predicate specifying a security policy.

Σ: A set of executions.

Π: A set of executions.

$\widehat{\mathcal{P}}$: A predicate on executions used in defining security policy $\mathcal{P}$.

σ: a finite or infinite execution.

σ': a finite execution.

τ: a finite or infinite execution.

τ': a finite execution.

$\sigma[..i]$: the prefix of σ involving its first i steps.

$\tau'\,\sigma$: denote finite execution τ' followed by execution σ.

Π^-: the set of all finite prefixes of elements in set Π.

Γ: A set of executions that is a safety property.

Q: The set of automaton states.

Q_0: The set of initial automaton states.

I: The set of automaton input symbols.

δ: The automaton next-state transition function.

Q': The current state of a security automaton.