Enforceable Security Policies*

Fred B. Schneider

Department of Computer Science
Cornell University
Ithaca, New York 14853

January 15, 1998

Abstract

A precise characterization is given for the class of security policies that can be enforced using mechanisms that work by monitoring system execution, and a class of automata is introduced for specifying those 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 things principals can infer about objects from observing other aspects of system behavior, or
- *availability*, and prohibit a principal from being denied use of a resource as a result of execution by other principals.

*Supported in part by ARPA/RADC grant F30602-96-1-0317 and AFOSR grant F49620-94-1-0198. The views and conclusions contained herein are those of the author 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.

To date, general-purpose security policies, like those above, have attracted most of the attention. But, application-dependent and special-purpose security policies are increasingly important. A system to support mobile code, like Java [5], 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 complicated rights-management schemes that restrict not only the use of stored materials but also the use of any derivatives [17].

The practicality of any security policy depends on whether that policy is enforceable and at what cost. In this paper, we address those questions for the class of enforcement mechanisms, which we call EM, that work by monitoring a *target system* and terminating any execution that is about to violate the security policy being enforced. Class EM includes security kernels, reference monitors, and all other operating system and hardware-based enforcement mechanisms that have appeared in the literature. Thus, understanding what can and cannot be accomplished using mechanisms in EM has value.

Excluded from EM are mechanisms that use more information than is available from monitoring the execution of a target system. Therefore, compilers and theorem-provers, which analyze a static representation of the target system to deduce information about all of its possible executions, are excluded from EM. The availability of information about all possible target system executions gives power to an enforcement mechanism—just how much power is an open question. Also excluded from EM are mechanisms that modify a target system before executing it. Presumably the modified target system would be "equivalent" to the original except that it satisfies the security policy of interest; a definition for "equivalent" is needed in order to analyze this class of mechanisms.

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.

2 Characteristics of EM Enforcement Mechanisms

Formally, we represent executions by finite and infinite sequences, where Ψ is the set of all possible such sequences.[1] Thus, a target system S defines a subset Σ_S of Ψ, and a *security policy* is a predicate on sets of executions. Target system S *satisfies* security policy $\mathcal{P}$ if and only if $\mathcal{P}(\Sigma_S)$ equals *true*.

Notice that, for 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 too many useful candidates. For instance, the requirement would preclude information flow (as defined informally in §1) from being considered a security policy—set 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) does not. In particular, when an execution is known to be in Π then the value of variable x reveals information about the value of y.

By definition, enforcement mechanisms in EM work by monitoring execution of the target system. Thus, any security policy $\mathcal{P}$ that can be enforced using a mechanism from EM must be equivalent to 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 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], a set of executions that can be defined by checking each execution individually is called a *property*. Using that terminology, we conclude from (1) that a security policy must characterize sets that are properties in order for the policy to have an enforcement mechanism in EM.

Not every security policy characterizes sets that are properties. Some predicates on sets of executions cannot be put in form (1) because they cannot be defined in terms of criteria that individual executions must each satisfy in isolation. For example, the information flow policy discussed above characterizes sets that are not properties (as is proved in [11]). Whether information flows from 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 characterize such sets of executions cannot be constructed only using predicates defined on single executions.

[1] 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.

Enforcement mechanisms in EM cannot base decisions on possible future execution, since that information is, by definition, not available to mechanisms in EM. This further restricts what security policies can be enforced by mechanisms from EM. In particular, consider security policy $\mathcal{P}$ of (1), and suppose τ is the prefix of some execution τ' where $\widehat{\mathcal{P}}(\tau) = \mathit{false}$ and $\widehat{\mathcal{P}}(\tau') = \mathit{true}$ hold. An enforcement mechanism for $\mathcal{P}$ must prohibit τ even though extension τ' satisfies $\widehat{\mathcal{P}}$, because otherwise execution of the target system might terminate before τ is extended into τ', and the enforcement mechanism would then have failed to enforce $\mathcal{P}$.

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. Then, the above requirement concerning execution prefixes violating $\widehat{\mathcal{P}}$, for security policy $\mathcal{P}$ defined by (1), is:

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

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 (2) and (3) are satisfied by sets that are *safety* properties [7], the class of properties that stipulate no "bad thing" happens during an execution. Formally, a property S is defined [8] to be a safety property if and only if for any finite or infinite execution σ,

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

holds. This means that S is a safety property if and only if S is characterized by a set of finite executions that are excluded and, therefore, the prefix of no execution in S. Clearly, a security policy $\mathcal{P}$ satisfying (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).

Our analysis of enforcement mechanisms in EM has established:

Unenforceable Security Policy: If the sets of executions characterized by a security policy $\mathcal{P}$ are not safety properties, then an enforcement mechanism from EM does not exist for $\mathcal{P}$.

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

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 invoked.
- 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 enforcement mechanisms in EM for exactly this policy.[2]
- Availability defines sets that are properties but not safety properties. In particular, any partial execution can be extended in a way that allows a principal to access a resource, so availability lacks a defining set of proscribed partial executions that every safety property must have. Thus, there are no enforcement mechanisms in EM for availability—at least as that policy is defined in §1.[3]

3 Security Automata

Enforcement mechanisms in EM work by terminating any target-system execution after seeing a finite prefix σ such that $\neg\widehat{\mathcal{P}}(\sigma)$ holds, for some predicate $\widehat{\mathcal{P}}$ defined by the policy being enforced. We established in §2 that the set of executions satisfying $\widehat{\mathcal{P}}$ also must be a safety property. Those being the only constraints on $\widehat{\mathcal{P}}$, we conclude that recognizers for sets of

[2]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. Given security policies $\mathcal{P}$ and $\mathcal{Q}$ for which $\mathcal{P} \Rightarrow \mathcal{Q}$ holds, a mechanism that enforces $\mathcal{P}$ does suffice for enforcing $\mathcal{Q}$. And, there do exist security policies that both imply restrictions on information flow and define sets that are safety properties. However, a policy $\mathcal{P}$ that implies $\mathcal{Q}$ might rule out executions that do not violate $\mathcal{Q}$, so using the stronger policy is not without adverse consequences.

[3]There are alternative formulations of availability that do characterize sets that are safety properties. An example is "one principal cannot be denied use of a resource for more than D steps as a result of execution by other principals". Here, the defining set of partial executions contains intervals that exceed D steps and during which a principal is denied use of a resource.

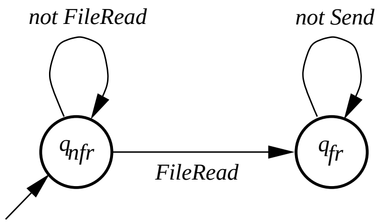

Figure 1: No $Send$ after $FileRead$

executions that are safety properties can serve as the basis for enforcement mechanisms in EM.

A class of automata for recognizing safety properties is defined (but not named) in [2]. We shall refer to these recognizers as *security automata*; they are similar to ordinary non-deterministic finite-state automata [6]. Formally, a security automaton is defined by:

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

I is dictated by the security policy being enforced; the symbols in I might correspond to system states, atomic actions, higher-level actions of the system, or state/action pairs. In addition, the symbols of I might also have to encode information about the past—for some safety properties, the transition function will require that information.

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

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

If ever Q'' is empty, 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.

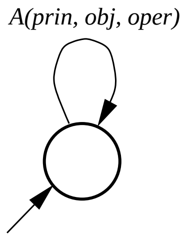

Figure 2: Access Control

Figure 1 depicts a security automaton for a security policy that prohibits execution of $Send$ operations after a $FileRead$ has been executed.. The automaton's 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 figure by unlabeled incoming edges, so automaton state q_{nfr} is the only initial automaton state. In the figure, transition function δ is specified in terms of edges labeled by *transition predicates*, which are 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 when Q' is the current state set, changes its current state set to

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

In Figure 1, transition predicate *not FileRead* is assumed to be satisfied by system execution steps that are not file read operations, and transition predicate *not Send* is assumed to be satisfied by system execution steps 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 if Figure 1 rejects inputs in which a message is sent after a file is read.

Another example of a security automaton is given in Figure 2. Different instantiations for transition predicate $A(prin, obj, oper)$ allow this automaton to specify either discretionary access control [9] or mandatory access control [3].

Discretionary Access Control. This policy prohibits operations according to an *access control matrix.* Specifically, given access control matrix M, a principal $Prin$ is permitted by the policy to execute an operation $Oper$ involving object Obj only if $Oper \in M[Prin, Obj]$ holds.

To specify this policy using the automaton of Figure 2, transition predicate $A(prin, obj, oper)$ would be instantiated by:

$$A(prin, obj, oper): \quad oper \in M[prin, obj]$$

Mandatory Access Control. This policy prohibits execution of operations according to a partially ordered set of *security labels* that are associated with system objects. Information in objects assigned higher labels is not permitted to be read and then stored into objects assigned lower labels.

For example, a system's objects might be assigned labels from the set

$$\{\text{topsecret},\ \text{secret},\ \text{sensitive},\ \text{unclassified}\}$$

ordered according to:

$$\text{topsecret} \succ \text{secret} \succ \text{sensitive} \succ \text{unclassified}$$

Suppose two system operations are supported—read and write. A mandatory access control policy might restrict execution of these operations according to:

(i) a principal p with label $\lambda(p)$ is permitted to execute $\mathsf{read}(F)$, which reads a file F with label $\lambda(F)$, only if $\lambda(p) \succeq \lambda(F)$ holds.

(ii) a principal p with label $\lambda(p)$ is permitted to execute $\mathsf{write}(F)$, which writes a file F with label $\lambda(F)$, only if $\lambda(F) \succeq \lambda(p)$ holds.

To specify this policy using the automaton of Figure 2, transition predicate $A(prin, obj, oper)$ is instantiated by:

$$\begin{array}{rll} A(prin, obj, oper): & & (oper = \mathsf{read} \ \wedge\ \lambda(prin) \succeq \lambda(obj)) \\ & \vee & (oper = \mathsf{write} \ \wedge\ \lambda(obj) \succeq \lambda(prin)) \\ & \vee & (oper \notin \{\mathsf{read}, \mathsf{write}\}) \end{array}$$

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

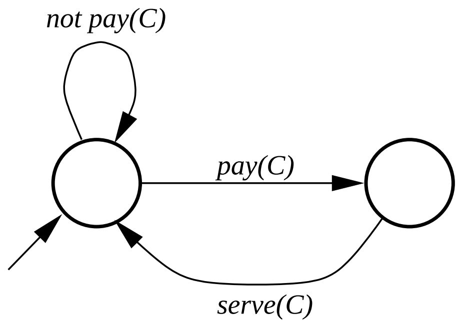

Figure 3: Security automaton for fair transaction

$pay(C)$:	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 operation that satisfies $pay(C)$. A security automaton for this policy is given in Figure 3.

Notice, the security automaton of Figure 3 does not stipulate that payment guarantees service. The security policy it specifies 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 do not impose 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—and therefore, according to §2, it is not enforceable using a mechanism from EM.

4 Using Security Automata for Enforcement

Any security automaton can serve as the basis for an enforcement mechanism in class EM, as follows. Each step the target system will next take is represented by an input symbol and sent to an implementation of the security automaton.

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

(ii) If the automaton cannot make a transition on that input symbol, then the target system is terminated.

In fact, any security policy enforceable using a mechanism from EM can be enforced using such a security-automaton implementation. This is because all safety properties have specifications as security automata, and Unenforceable Security Policy of §2 implies that EM enforcement mechanisms enforce safety properties. Consequently, by understanding the limitations of security-automata enforcement mechanisms, we can gain insight into the limitations of all enforcement mechanisms in class EM.

Implicit in (ii) is the assumption that the target system can be terminated by the enforcement mechanism. Specifically, we assume that the enforcement mechanism has sufficient control over the target system 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, consider the following variation on availability of §1:

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

Sets satisfying Real-Time Availability are safety properties—the "bad thing" is an interval of execution spanning more than D seconds during which some principal is denied the resource. The input symbols of a security automaton for Real-Time Availability must therefore encode time. However, the passage of time cannot be stopped, so a target system with real-time clocks cannot be prevented from continuing to produce input symbols. Real-Time Availability simply cannot be enforced using one of our automata-based enforcement mechanisms, because target systems lack the necessary controls. 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.

Two mechanisms are involved in our security-automaton implementation of an enforcement mechanism.

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

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

Their aggregate cost can be quite high. For example, when the automaton's input symbols are the set of program states and its transition predicates are arbitrary state predicates, a new input symbol is produced for each machine-language instruction that the target system executes. The enforcement mechanism must be invoked before every target-system instruction.

However, for security policies where the target system's production of input symbols coincides with occurrences of hardware traps, our 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 system-supported objects, like files. Here, the target system's production of input symbols coincides with invocations of system operations, hence the production of input symbols coincides with occurrences of system-call traps.

A second example of exploiting hardware traps arises in implementing memory protection. Memory protection implements discretionary access control with operations read, write, and execute and an access control matrix that tells how processes can access each region of memory. The security automaton of Figure 2 specifies this security policy. Notice that this security automaton expects an input symbol for each memory reference. But most, if not all, of the 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 system'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.

Inexpensive implementation of our automata-based enforcement mechanisms is also possible when programs are executed by a software-implemented virtual machine (sometimes known as a reference monitor). 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[10] could easily be augmented to implement the Automaton Input Read and Automaton Transition mechanisms for input symbols that correspond to method invocations.

Beyond Class EM Enforcement Mechanisms

The overhead of enforcement can sometimes be reduced by merging the enforcement mechanism into the target system. One such scheme, which has recently attracted attention, is *software-based fault isolation* (SFI), also known as "sandboxing" [19, 16]. SFI implements memory protection, as specified by a one-state automaton like that of Figure 2, but does so without hardware assistance. Instead, a program is edited before it is executed, and only such edited programs are run by the target system. (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 class EM because SFI involves modifying the target system, and modifications are not permitted for 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 system instruction that produces an input symbol. Notice that nothing prevents the SFI approach from being used with multi-state automata, thereby enforcing any security policy that can be specified as a security automaton. Moreover, a program optimizer should be able to simplify the inserted code and eliminate useless portions[4] although this introduces a second type of program analysis to SFI and requires putting further trust in automated program analysis tools.

Finally, there is no need for any run-time enforcement mechanism if the target system 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—a policy specified by one-state security automata—in *proof carrying code* (PCC) [13]. 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

[4] Úlfar Erlingsson of Cornell has implemented a system that does exactly this for the Java virtual machine and for Intel x86 machine language.

certain security policies specified by one-state security automata, a compiler can automatically produce PCC from programs written in high-level, type-safe programming languages[12, 14].

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 [2].

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 system. 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 system. 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 Buchi automata [4], and Buchi automata 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 some 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.

The designer of a security automaton often must choose between encoding security-relevant information in the target system's state and in an automaton state. Larger automata are usually more complicated, hence more difficult to understand, and often lead to more expensive enforcement mechanisms. For example, our generalization of SFI involves modifying the target system by inserting code and then employing a program optimizer to simplify the result. The inserted code simulates the security automaton, and the code for a smaller security automaton will be smaller, cheaper to execute, and easier to optimize. Similarly, for our generalization of PCC, proof obligations derived according to [2] are fewer and simpler if the security automata is smaller.

However, we conjecture that the cost of executing a reference monitor that implements Automaton Transition is probably insensitive to whether security-relevant state information is stored in the target system or in automaton states. This is because the predicates that must be evaluated in the two implementations will differ only in whether a state component has been associated with the security automaton or the target system, and the cost of reading that state information and of evaluating the predicates should be similar.

Application to Safety-Critical Systems

The idea that security kernels might have application in safety-critical systems is eloquently justified in [15] and continues to interest researchers [18]. Safety-critical systems are concerned with enforcing properties that are safety properties (in the sense of [8]), 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 [15] 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, [15] concludes that safety properties but not liveness properties are enforceable. However, the arguments given in [15] 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.

In addition, because [15] 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 [15]. 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 [15], 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 an earlier draft of this paper were also provided by Earl Boebert, Li Gong, Robert Grimm, Keith Marzullo, and John Rushby. John McLean served as a valuable sounding board for these ideas as I developed them. Feedback from Martin Abadi helped to sharpen the formalism. 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] Bell, D.E. and L.J. La Padula. Secure computer systems: Mathematical foundations. Technical Report ESD-TR-73-278, Hanscom AFB, Bedford, Mass., Nov. 1973.

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

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

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

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

[8] 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.

[9] 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.

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

[11] 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.

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

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

[14] Necula, G.C. amd P. Lee. The design and implementation of a certifying compiler. Submitted for publication, 1998.

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

[16] 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.

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

[18] 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.

[19] 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.