Chapter 11

Reference Monitors

Isolation may block system access by attackers, but it also blocks access by legitimate users. So it makes sense to consider less-draconian defenses. A *reference monitor* is one such defense. It intercepts invocations of certain actions, blocking any invocation that, if allowed to proceed, would violate the security policy being enforced. Reference monitors originally were conceived as mechanisms for ensuring that accesses to some class of objects would comply with a given authorization policy. But reference monitors can be used to enforce a much broader class of security policies, which are defined using predicates to characterize sets of states from which an operation should be allowed to proceed.

11.1 Reference Monitor Requirements

A reference monitor is invoked with each *monitored access* attempted by an *untrusted principal* whose compliance with a given security policy is sought. The untrusted principals might correspond to user sessions, processes, or specific programs. When a monitored access causes a reference monitor to be invoked, code we call the *monitor response* is executed to determine whether allowing that monitored access to proceed would comply with the security policy being enforced. If performing the monitored access would not comply, then execution of the invoker is blocked; otherwise, execution of the invoker is allowed to continue. To block execution, a monitor response often will terminate execution of the untrusted principal that is attempting the access.

When a reference monitor is used for validating accesses to memory or to files, the untrusted principals are processes and the monitored accesses are read, write, and execute operations. The monitor response allows an invoking process to continue executing only if that process is executing for a user holding the appropriate privileges for the requested access. An operating system that supports access control lists or capabilities is implementing such a reference monitor, as is a processor that supports virtual memory having segment descriptors that specify whether read, write, or execute access is permitted.

Reference monitors can be used to enforce a wide variety of other security policies if monitor responses (i) update some *monitor state* as part of a monitor response and (ii) block further execution by the invoker of a monitored access whenever the arguments to that monitored access in conjunction with the current monitor state do not satisfy some given predicate. So the security policy being enforced is based on an arbitrary predicate rather than being limited to checking privileges held by the invoking untrusted principal.

For a reference monitor to be trustworthy, we must have assurance that it will satisfy our expectations about which executions it allows. We facilitate that trustworthiness if a reference monitor and the environment in which it is deployed satisfy the following requirements.

> *Complete Mediation.* The reference monitor is invoked whenever a monitored access is attempted.
>
> *Tamperproof.* Other execution cannot interfere with monitor responses or alter the monitor state.
>
> *Assurance.* The code that implements a monitor's responses and implements the mechanisms for invoking that code when any monitored access is attempted is small and simple. Therefore, testing and/or analysis can provide assurance that the Complete Mediation and Tamperproof requirements hold.

For purposes of satisfying these requirements, reference monitor implementations traditionally have considered untrusted principals to be opaque. So support for a reference monitor typically does alter the untrusted principals or use knowledge about their internals.[1]

11.2 Classical Implementation

All execution is performed by interpreters that are implemented in hardware and/or in software. Such interpreters include the CPU, a hypervisor, the operating system, middleware, and various runtime libraries. Complete mediation for a reference monitor thus can be seen as implementing extensions to an interpreter. All principals executed by the interpreter become the untrusted principals for the reference monitor that the interpreter extensions are implementing. Notice, no changes are being made to those untrusted principals and no information is required about their internals.

An interpreter that supports extensions will have a set E_{Int} of events, where the occurence of an event in E_{Int} immediately causes the interpreter to begin executing a *handler* that has been provided for that event. Low-overhead implementations of complete mediation for a reference monitor then become possible if each monitored access corresponds to an element of E_{Int}. The handlers provided for events in E_{Int} would either serve as the monitor responses or transfer

[1] A reference monitor implementation that does analyze and modify the untrusted principals is discussed in §12.4.

control to other code that implements the monitor responses. For an even lower-overhead implementation, a monitored access that should be allowed to proceed and does not update monitored state would not be associated with elements in E_{Int} and, thus, its execution would not be interrupted.

Interpreters implemented by hardware CPUs instantiate this architecture when they transfer control to an associated *trap handler* in response to the following *traps*.

- Execution of a system call instruction.

- Access to a virtual memory segment where the segment descriptor indicates that the segment is not present or the segment does not permit the kind of access (read, write, or execute) being attempted.

Given such an instance of this architecture, we implement a reference monitor using these traps as the monitored accesses. It is no accident that these traps are ideal for implementing a reference monitor to prevent abuses of operating system abstractions and services—processes request operating system services either by making sytem calls or by performing memory-mapped I/O. Also, notice that changes to the segment table can be used to avoid the occurence (and overhead) of traps for those monitored accesses that (i) should be permitted to proceed and (ii) do not need to update monitor state.

Interpreters implemented in software can be designed to invoke a specified handler whenever the interpreter state starts satisfying some given predicate. The code for monitoring such a predicate might be added when the interpreter is being written, as a patch after the interpreter has been deployed, or using an interface that allows executing clients to add predicates for monitoring. Of course, monitoring the value of a predicate slows an interpreter if the predicate must be evaluated frequently, if evaluating the predicate is expensive due to the computation involved, or if a large amount of state needs to be read for evaluating the predicate.

A reference monitor is often made tamperproof by placing it in an operating system kernel, so the isolation mechanism that is protecting the kernel also isolates the execution of monitor responses and protects the monitor state. Being part of a kernel also gives monitor responses ready access to the operations that a kernel typically will support to block further execution of a process. The assurance requirement for a reference monitor becomes difficult to satisfy, though, when a kernel is large and complicated. That is why a security kernel is, by definition, small and simple, yet provides the functionality needed to satisfy the tamperproof and complete mediation requirements for implementing a reference monitor.[2]

[2]Security kernels are described on page 306 in §10.2.3 about kernel functionality.

11.3 *Enforceable Policies and EM

To characterize the class of security policies that can be enforced using reference monitors, define Σ_S to be the set of execution traces corresponding to possible executions for a program S, and define $Enf_{\mathcal{R}}(\Sigma_S)$ to be the set of execution traces for executions of S when reference monitor $\mathcal{R}$ is present. For security policies specified using predicates $P(\cdot)$ on sets of execution traces, a reference monitor $\mathcal{R}$ *enforces* security policy $P(\cdot)$ if $P(Enf_{\mathcal{R}}(\Sigma_S))$ holds for every program S.

The analysis given below assumes that monitor responses implement computable functions, but it ignores the execution time for monitor responses, the size of the monitor's state, and the overhead for implementing complete mediation. The resulting characterization of security policies enforceable using reference monitors is thus independent of algorithm design, the choice of programming language, and the run-time environment. So when the characterization says that a security policy $P(\cdot)$ cannot be enforced by any reference monitor, then some other type of enforcement mechanism must be used to enforce $P(\cdot)$. Because our analysis is ignoring overheads, though, the characterization might say that a reference monitor can enforce a given security policy even if the resulting performance would be unacceptable.

To derive our characterization of what security policies can be enforced by reference monitors, we proceed as follows.

(a) We define the class EM (for Execution Monitoring) of reference monitors.

(b) We show that for every reference monitor $\mathcal{R}$, there is a corresponding reference monitor $\hat{\mathcal{R}}$ that is in EM and that enforces the same security policies as $\mathcal{R}$ does.

(c) We give a formal definition for the set $Enf_{\hat{\mathcal{R}}}(\Sigma_S)$ of execution traces for a program S that is executed in the presence of an EM reference monitor $\hat{\mathcal{R}}$.

(d) We derive a characterization for security policies that reference monitors in EM can enforce and discuss why certain policies cannot be enforced.

The conclusion of step (b), implies that the characterization for EM-enforceable policies obtained in step (d) is also a characterization of policies enforceable by reference monitors not in EM.

Definition of EM. All reference monitors in class EM follow a template that facilitates characterizing the policies those reference monitors enforce.

> EM **Template.** Let $\hat{\mathcal{R}}$ be a reference monitor implemented as extensions to an interpreter $\mathcal{I}$ that is executing a program S for some untrusted principal. $\hat{\mathcal{R}}$ is in class EM if it is defined in terms of
>
> – a variable $hist_{\hat{\mathcal{R}}}$ that stores a finite sequence of interpreter states,
>
> – a predicate $p_{\hat{\mathcal{R}}}(hist_{\hat{\mathcal{R}}})$ that is deterministic, computable, and satisfies $p_{\hat{\mathcal{R}}}(\epsilon) = true$, and

– a monitor response $\hat{\mathcal{R}}$ that, for each monitored access, proceeds as follows:

(i) $\hat{\mathcal{R}}$ appends to $hist_{\hat{\mathcal{R}}}$ the current state s of interpreter $\mathcal{I}$ (which by construction would include the state of S) and then

(ii) $\hat{\mathcal{R}}$ blocks further execution of S if $\neg p_{\hat{\mathcal{R}}}(hist_{\hat{\mathcal{R}}})$ holds; otherwise, execution of S is allowed to continue. □

So every EM reference monitor $\hat{\mathcal{R}}$ is maintaining in $hist_{\hat{\mathcal{R}}}$ a subsequence[3] of the interpreter states. That subsequence in $hist_{\hat{\mathcal{R}}}$ is the list containing the interpreter state that existed each time a monitored access was triggered by S.

Construction of $\hat{\mathcal{R}}$ from $\mathcal{R}$. A reference monitor $\mathcal{R}$ in a real system is unlikely to follow the EM Template.

- Instead of using a single variable $hist_{\mathcal{R}}$ that grows without bound, the monitor response for $\mathcal{R}$ would store the monitor state in a collection of bounded-size *monitor variables*. Updates to monitor variables would be calculated from (i) their current values, and (ii) the current interpreter state (including the state of program S).
- Instead of using a single predicate $p_{\mathcal{R}}(hist_{\mathcal{R}})$ to determine whether further execution by S should be blocked, the monitor response for $\mathcal{R}$ might use multiple predicates, checking the different predicates under different circumstances, such as different operation attempts by S.

We now show how to transform such a reference monitor $\mathcal{R}$ into an EM reference monitor $\hat{\mathcal{R}}$ that enforces the same policies.

The first step is to restructure $\mathcal{R}$—perhaps by adding monitor variables—so that the decision to block further execution of S is made by checking a single predicate $p_{\mathcal{R}}(\cdot)$ once and at the end of the monitor response.[4] The restructured monitor response would have the following form, where $\bar{v}$ denotes the list of monitor variables, $f_{\mathcal{R}}(\bar{v}, s)$ evaluates to a list of the new values to store in the monitor variables, and s is the state of the interpreter (and, thus includes the state of S) when the monitored access was attempted.

$$\bar{v} := f_{\mathcal{R}}(\bar{v}, s);\ \textbf{if } \neg p_{\mathcal{R}}(\bar{v}) \textbf{ then} \text{ block further execution of } S \tag{11.1}$$

Function $f_{\mathcal{R}}(\cdot,\cdot)$ is guaranteed to exist because we assumed that (i) a monitor response implements a computable function of the values it reads, and (ii) monitor variables and interpreter state are the only values that a monitor response reads.

For the next step of the transformation, we eliminate the use of monitor variables. Values stored by the monitor variables $\bar{v}$ just before (11.1) checks

[3] Recall, σ is considered a subsequence of a sequence τ if σ can be obtained by deleting some of the elements in τ.

[4] A detailed account of the transformation would be specific to the programming language used to code the monitor response. We omit the account here, because there is little to learn from seeing the transformations.

predicate $\neg p_{\mathcal{R}}(\bar{v})$ are computable from initial values $\bar{v}_0$ of $\bar{v}$, the current interpreter state, and the interpreter states read during previously executed monitor responses. Since $hist_{\hat{\mathcal{R}}}$ is a sequence containing the state of the interpreter at the start of each previously executed monitor response, we have that

- final state $\mathsf{last}(hist_{\hat{\mathcal{R}}})$ of $hist_{\hat{\mathcal{R}}}$ is the interpreter's state when the current monitor response started executing, and
- prefix $hist_{\hat{\mathcal{R}}}[1..|hist_{\hat{\mathcal{R}}}|-1]$ containing all but the final state of $hist_{\hat{\mathcal{R}}}$ is what determined the value of $\bar{v}$ at the start of the current monitor response.

So a (recursive) function $F_{\hat{\mathcal{R}}}(\bar{v}_0, hist_{\hat{\mathcal{R}}})$ for computing that value of $\bar{v}$ can be constructed from $f_{\mathcal{R}}(\cdot,\cdot)$ as follows.

$$F_{\hat{\mathcal{R}}}(\bar{v}_0, hist_{\hat{\mathcal{R}}}){:}\ \begin{cases} f_{\mathcal{R}}(\bar{v}_0, \mathsf{last}(hist_{\hat{\mathcal{R}}})) & \text{if } |hist_{\hat{\mathcal{R}}}| = 1 \\ f_{\mathcal{R}}(F_{\hat{\mathcal{R}}}(\bar{v}_0, hist_{\hat{\mathcal{R}}}[1..|hist_{\hat{\mathcal{R}}}|-1]), \mathsf{last}(hist_{\hat{\mathcal{R}}})) & \text{if } |hist_{\hat{\mathcal{R}}}| > 1 \end{cases}$$

By replacing $\bar{v}$ in $\neg p_{\mathcal{R}}(\bar{v})$ of (11.1) with $F_{\hat{\mathcal{R}}}(\bar{v}_0, hist_{\hat{\mathcal{R}}})$, we obtain a monitor response for a reference monitor $\hat{\mathcal{R}}$ that enforces the same policies as reference monitor $\mathcal{R}$ but follows the EM Template. Specifically, we define $p_{\hat{\mathcal{R}}}(hist_{\hat{\mathcal{R}}})$ to be $p_{\mathcal{R}}(F_{\hat{\mathcal{R}}}(\bar{v}_0, hist_{\hat{\mathcal{R}}}))$ to get the following instance of the EM Template for reference monitor $\hat{\mathcal{R}}$.

$$\begin{array}{l} hist_{\hat{\mathcal{R}}} := hist_{\hat{\mathcal{R}}} \cdot s; \\ \textbf{if } \neg p_{\hat{\mathcal{R}}}(hist_{\hat{\mathcal{R}}}) \textbf{ then} \text{ block further execution of } S \end{array} \tag{11.2}$$

11.3.1 EM-Enforceable Policies

A security policy $P(\cdot)$ is defined to be EM-*enforceable* when there is an EM reference monitor $\hat{\mathcal{R}}$ that causes $P(Enf_{\hat{\mathcal{R}}}(\Sigma_S))$ to hold for all programs S. Each execution trace $\sigma \in \Sigma_S$ is a (possibly infinite) sequence $s_0\, s_1\, s_2 \ldots$ of states, where every state in the sequence contains all of the information needed to generate the next state in the sequence. So if the interpreter is a hardware CPU, then the state would include a program counter that identifies which atomic action produces the next state. The state also includes the current time.

When a reference monitor $\mathcal{R}$ is present, certain states will cause a monitored access followed by execution of a monitor response. For an execution trace σ, we write $\mathsf{ma}_{\mathcal{R}}(\sigma)$ to denote the subsequence of σ comprised of those states that caused monitored accesses. So an execution trace σ corresponding to an execution of S when a reference monitor $\hat{\mathcal{R}}$ described by (11.2) is present can be described as follows.

(i) $\hat{\mathcal{R}}$ checks whether $\neg p_{\hat{\mathcal{R}}}(hist_{\hat{\mathcal{R}}})$ holds when each of the successively longer prefixes of $\mathsf{ma}_{\hat{\mathcal{R}}}(\sigma)$ is stored in $hist_{\hat{\mathcal{R}}}$.

(ii) If $\hat{\mathcal{R}}$ ever observes a prefix ω of $\mathsf{ma}_{\mathcal{R}}(\sigma)$ where $\neg p_{\hat{\mathcal{R}}}(\omega)$ holds, then $\hat{\mathcal{R}}$ blocks further execution of S.

$Enf_{\hat{\mathcal{R}}}(\Sigma_S)$ thus can be formalized as the union of a set $Norm_{\hat{\mathcal{R}}}(\Sigma_S)$ of execution traces for normal executions and a set $Term_{\hat{\mathcal{R}}}(\Sigma_S)$ of the execution traces for executions blocked by $\hat{\mathcal{R}}$.

$$Enf_{\hat{\mathcal{R}}}(\Sigma): \quad Norm_{\hat{\mathcal{R}}}(\Sigma) \cup Term_{\hat{\mathcal{R}}}(\Sigma) \tag{11.3}$$

$Norm_{\hat{\mathcal{R}}}(\Sigma_S)$ is the subset of Σ_S corresponding to executions of S that were not blocked by $\hat{\mathcal{R}}$. So if $\sigma \in Norm_{\hat{\mathcal{R}}}(\Sigma_S)$ holds then $\sigma \in \Sigma_S$ holds and $p_{\hat{\mathcal{R}}}(hist_{\hat{\mathcal{R}}})$ holds each time a monitor response is invoked. Those values of $hist_{\hat{\mathcal{R}}}$ are the prefixes of $\mathsf{ma}_{\hat{\mathcal{R}}}(\sigma)$. Writing $\alpha \preceq \beta$ to denote that α is a prefix of β, instances of $p_{\hat{\mathcal{R}}}(hist_{\hat{\mathcal{R}}})$ that $\hat{\mathcal{R}}$ evaluates are together equivalent to $p^*_{\hat{\mathcal{R}}}(hist_{\hat{\mathcal{R}}})$ defined by:

$$p^*_{\hat{\mathcal{R}}}(\sigma): \quad (\forall \omega \preceq \mathsf{ma}_{\hat{\mathcal{R}}}(\sigma): \ p_{\hat{\mathcal{R}}}(\omega)) \tag{11.4}$$

Thus, we get the following definition for $Norm_{\hat{\mathcal{R}}}(\Sigma_S)$:

$$Norm_{\hat{\mathcal{R}}}(\Sigma_S): \quad \{\sigma \mid \sigma \in \Sigma_S \ \wedge \ p^*_{\hat{\mathcal{R}}}(\sigma)\} \tag{11.5}$$

$Term_{\hat{\mathcal{R}}}(\Sigma_S)$ adds to $Enf_{\hat{\mathcal{R}}}(\Sigma)$ execution traces for executions that $\hat{\mathcal{R}}$ blocked. If $\sigma \in \Sigma_S$ and $\neg p^*_{\hat{\mathcal{R}}}(\sigma)$ hold then $\sigma \notin Norm_{\hat{\mathcal{R}}}(\Sigma_S)$ holds and $Term_{\hat{\mathcal{R}}}(\Sigma_S)$ contains the longest prefix τ of σ that satisfies $p^*_{\hat{\mathcal{R}}}(\tau)$. The following formal definition for $Term_{\hat{\mathcal{R}}}(\Sigma_S)$ uses notation $\beta^{[-i]}$ to denote the prefix obtained by deleting the last i states from finite sequence β.

$$\begin{aligned} Term_{\hat{\mathcal{R}}}(\Sigma_S): \quad \{\tau \mid \ & \sigma \in \Sigma_S \wedge \\ & (\exists \beta \preceq \sigma: \ \neg p_{\hat{\mathcal{R}}}(\mathsf{ma}_{\hat{\mathcal{R}}}(\beta)) \ \wedge \ p^*_{\hat{\mathcal{R}}}(\beta^{[-1]}) \ \wedge \ \tau = \beta^{[-1]})\} \end{aligned} \tag{11.6}$$

The security policy enforced by an EM reference monitor $\hat{\mathcal{R}}$ is specified by a predicate $P_{\hat{\mathcal{R}}}(\Sigma)$ that is *true* if and only if no execution trace in Σ corresponds to an execution that is blocked by $\hat{\mathcal{R}}$:

$$P_{\hat{\mathcal{R}}}(\Sigma): \quad \Sigma = Norm_{\hat{\mathcal{R}}}(\Sigma) \tag{11.7}$$

We argue that definition (11.7) for $P_{\hat{\mathcal{R}}}(\cdot)$ is sensible by proving (11.7) satisfies two properties that we expect to hold for a definition of $P_{\hat{\mathcal{R}}}(\cdot)$.

The first property is that an EM reference monitor $\hat{\mathcal{R}}$ does not block any execution corresponding to an execution trace from a set Σ where $P_{\hat{\mathcal{R}}}(\Sigma)$ holds, since having $P_{\hat{\mathcal{R}}}(\Sigma)$ hold should imply that Σ satisfies the policy $\hat{\mathcal{R}}$ is enforcing. That is formalized as

$$P_{\hat{\mathcal{R}}}(\Sigma) \Rightarrow (\Sigma = Enf_{\hat{\mathcal{R}}}(\Sigma)),$$

and it is proved by using the observation that the existential forming the second conjunct in definition (11.6) is *false* when execution trace $\sigma \in \Sigma$ satisfies $p^*_{\hat{\mathcal{R}}}(\sigma)$,

so $Term_{\hat{\mathcal{R}}}(\Sigma) = \varnothing$ holds when all elements $\sigma \in \Sigma$ satisfy $p^*_{\hat{\mathcal{R}}}(\sigma)$:

$$
\begin{array}{ll}
 & P_{\hat{\mathcal{R}}}(\Sigma) \\
\Rightarrow & \text{(11.7)} \\
 & Norm_{\hat{\mathcal{R}}}(\Sigma) = \Sigma \\
\Rightarrow & \text{definition (11.3) of } Enf_{\hat{\mathcal{R}}}(\Sigma) \text{ and } Term_{\hat{\mathcal{R}}}(\Sigma) = \varnothing \\
 & Enf_{\hat{\mathcal{R}}}(\Sigma) = \Sigma
\end{array}
$$

The second property is that $P_{\hat{\mathcal{R}}}(Enf_{\hat{\mathcal{R}}}(\Sigma))$ holds, indicating that $P_{\hat{\mathcal{R}}}(\cdot)$ is satisfied by the set of execution traces corresponding to execution when EM reference monitor $\hat{\mathcal{R}}$ is present. The proof hinges on the definitions of $Norm_{\hat{\mathcal{R}}}(\Sigma)$ and $Term_{\hat{\mathcal{R}}}(\Sigma)$—specifically, that if $\sigma \in Norm_{\hat{\mathcal{R}}}(\Sigma)$ or $\sigma \in Term_{\hat{\mathcal{R}}}(\Sigma)$ holds then $p^*_{\hat{\mathcal{R}}}(\sigma)$ holds. According to definition (11.7) of $P_{\hat{\mathcal{R}}}(\cdot)$, a proof of $P_{\hat{\mathcal{R}}}(Enf_{\hat{\mathcal{R}}}(\Sigma))$ requires proving:

$$Enf_{\hat{\mathcal{R}}}(\Sigma) = Norm_{\hat{\mathcal{R}}}(Enf_{\hat{\mathcal{R}}}(\Sigma)).$$

Here is a sketch of that proof.

$$
\begin{array}{ll}
 & Enf_{\hat{\mathcal{R}}}(\Sigma) \\
= & \text{definition (11.3) of } Enf_{\hat{\mathcal{R}}}(\Sigma) \\
 & Norm_{\hat{\mathcal{R}}}(\Sigma) \ \cup \ Term_{\hat{\mathcal{R}}}(\Sigma) \\
= & \text{definition (11.6) of } Term_{\hat{\mathcal{R}}}(\Sigma)\text{: } \sigma \in Term_{\hat{\mathcal{R}}}(\Sigma) \Rightarrow p^*_{\hat{\mathcal{R}}}(\sigma) \\
 & Norm_{\hat{\mathcal{R}}}(\Sigma) \ \cup \ \{\sigma \mid \sigma \in Term_{\hat{\mathcal{R}}}(\Sigma) \ \wedge \ p^*_{\hat{\mathcal{R}}}(\sigma)\} \\
= & \text{definition (11.5) of } Norm_{\hat{\mathcal{R}}}(\Sigma)\text{: } \sigma \in Norm_{\hat{\mathcal{R}}}(\Sigma) \Rightarrow p^*_{\hat{\mathcal{R}}}(\sigma) \\
 & \{\sigma \mid \sigma \in Norm_{\hat{\mathcal{R}}}(\Sigma) \ \wedge \ p^*_{\hat{\mathcal{R}}}(\sigma)\} \ \cup \ \{\sigma \mid \sigma \in Term_{\hat{\mathcal{R}}}(\Sigma) \ \wedge \ p^*_{\hat{\mathcal{R}}}(\sigma)\} \\
= & \text{set theory} \\
 & \{\sigma \mid \sigma \in Norm_{\hat{\mathcal{R}}}(\Sigma) \cup Term_{\hat{\mathcal{R}}}(\Sigma) \ \wedge \ p^*_{\hat{\mathcal{R}}}(\sigma)\} \\
= & \text{definition (11.3) of } Enf_{\hat{\mathcal{R}}}(\Sigma) \\
 & \{\sigma \mid \sigma \in Enf_{\hat{\mathcal{R}}}(\Sigma) \ \wedge \ p^*_{\hat{\mathcal{R}}}(\sigma)\} \\
= & \text{definition (11.5) of } Norm_{\hat{\mathcal{R}}}(\Sigma) \\
 & Norm_{\hat{\mathcal{R}}}(Enf_{\hat{\mathcal{R}}}(\Sigma))
\end{array}
$$

11.3.2 Policies that Reference Monitors Cannot Enforce

Definition (11.3) of $Enf_{\hat{\mathcal{R}}}(\cdot)$ is a predicate that will be satisfied by the set of execution traces corresponding to executions that an EM reference monitor $\hat{\mathcal{R}}$ allows. The formulation of that predicate gives insights into characteristics of EM-enforceable security policies. Moreover, since every reference monitors enforces the same security policy as some EM reference monitor, the insights about EM-enforceable security policies also apply to the security policies that any other reference monitor enforces. The text below explores limitations that are implied by those properties.

Limitation: Executions Checked in Isolation. Expanding definition (11.3) for $Enf_{\hat{\mathcal{R}}}(\cdot)$ reveals that whether $\sigma \in Enf_{\hat{\mathcal{R}}}(\Sigma)$ holds for execution trace σ depends only on σ and not on other execution traces in Σ. This conclusion should not be surprising, since a reference monitor blocks an execution based only on the states that reference monitor observes during that execution.

So for a security policy $P(\cdot)$ to be EM-enforceable, there will exist a predicate $p(\cdot)$ on execution traces, where $p(\sigma)$ holds when execution trace σ corresponds to an execution that complies with $P(\cdot)$ and $p(\cdot)$ does not depend on other execution traces. Thus, a security policy $P(\cdot)$ is not EM-enforceable unless it is equivalent to

$$P(\Sigma)\colon \quad (\forall \sigma \in \Sigma\colon\ p(\sigma)) \tag{11.8}$$

for some predicate $p(\cdot)$ on execution traces.

The requirement that a predicate $P(\cdot)$ specifying an EM-enforceable security policy be expressible in canonical form (11.8) limits what policies are EM-enforceable. For some sets Σ, the membership of each element is determined by the presence or attributes of other members in that same set. Predicates having form (11.8) cannot characterize such sets, because $p(\cdot)$ has a single argument and, thus, cannot depend on the other elements of Σ. So there will be security policies that cannot be specified using a predicate having form (11.8). Those security policies will not be EM-enforceable.

Expressiveness limits arising from the form of (11.8) matter only if there are inexpressible policies that we might want to enforce by using a reference monitor. Unfortunately, there could be. An example is the security policy requiring that the initial value of a secret variable x has no effect on the final value of a public variable y in a deterministic system. A predicate to specify this[5] security policy would be satisfied by any set of execution traces Σ in which the final states in each pair of execution traces has the same value for variable y whenever the initial states in the pair differ only in the value of variable x. Define $init(x, \sigma, \sigma')$ to be a predicate that is satisfied if the initial states of execution traces σ and σ' differ only in the value of variable x, and define $fin(y, \sigma, \sigma')$ to be a predicate that is satisfied if final states of execution traces σ and σ' agree on the value for variable y. Then a predicate $NE(\cdot)$ (for no effect) to restrict the value of x from having an effect on y can be specified as follows.

$$NE(\Sigma)\colon \quad (\forall \sigma, \sigma' \in \Sigma\colon\ init(x, \sigma, \sigma') \Rightarrow fin(y, \sigma, \sigma'))$$

To put $NE(\Sigma)$ into the same form as (11.8) would require defining a predicate $p(\cdot)$. Only a predicate that depends on Σ could determine whether an execution trace σ is satisfying $NE(\Sigma)$. But having Σ be an argument to $p(\cdot)$ is not permitted by the form of (11.8). So an EM reference monitor cannot be used to enforce $NE(\cdot)$, which means no reference monitor can be used to enforce this security policy.

[5] Chapter 9 has a detailed discussion of information flow security policies.

Limitation: Violations are Prefixes. $\hat{\mathcal{R}}$ excludes from $Enf_{\hat{\mathcal{R}}}(\Sigma)$ any execution trace having a finite prefix ω where $\neg p_{\hat{\mathcal{R}}}(\mathsf{ma}_{\hat{\mathcal{R}}}(\omega))$ holds. Thus, to be EM-enforceable, a security policy $P(\cdot)$ must be excluding those execution traces σ having a finite prefix where some predicate (say) $viol_P(\cdot)$ holds. That means predicate $p(\cdot)$ in (11.8) must satisfy

$$p(\sigma) \;\Rightarrow\; \neg(\exists \omega \preceq \sigma\colon\; viol_P(\omega)).$$

The requirement that violations are prefixes means that an EM reference monitor cannot enforce a security policy that requires one or more actions in the indefinite future. As an example, consider a security policy that specifies availability: every request is eventually serviced. A predicate $p_{\hat{\mathcal{R}}}(\cdot)$ defined on prefixes of execution traces cannot detect the security violation being proscribed by this security policy, because any prefix containing an unserviced request still might be extended to a prefix where that request has been serviced. So an EM reference monitor would have no basis to block non-compliant executions; the security policy is not EM-enforceable.

Limitation: Granularity of Monitored Accesses. The frequency of monitored accesses limits the information $\hat{\mathcal{R}}$ has for deciding whether to block further execution, in turn limiting the policies that can be enforced by an EM reference monitor. In particular, the predicate $p_{\hat{\mathcal{R}}}(\cdot)$ used by an EM reference monitor $\hat{\mathcal{R}}$ (11.2) to decide that further execution should be blocked must satisfy

$$viol_{\mathcal{P}}(\omega) \Rightarrow \neg p_{\hat{\mathcal{R}}}(\mathsf{ma}_{\hat{\mathcal{R}}}(\omega)) \tag{11.9}$$

for all prefixes ω of execution traces, since $\hat{\mathcal{R}}$ blocks further execution of S only if a prefix ω is reached that satisfies $\neg p_{\hat{\mathcal{R}}}(\mathsf{ma}_{\hat{\mathcal{R}}}(\omega))$. Notice, if the prefix ω that first makes $viol_{\mathcal{P}}(\omega)$ become *true* does not end with a monitored access then (11.9) requires $\neg p_{\hat{\mathcal{R}}}(\mathsf{ma}_{\hat{\mathcal{R}}}(\omega^{[-i]}))$ to hold for some i where $0 < i$, enabling the reference monitor to anticipate $viol_{\mathcal{P}}(\cdot)$ first becoming *true*.

Predicates that anticipate violations can lead to unpleasantly conservative enforcement, though. For example, $\omega^{[-i]}$ discussed above might be a prefix both of an execution trace that has a violation and an execution trace that does not have a violation. In that case, a reference monitor that uses $\neg p_{\hat{\mathcal{R}}}(\mathsf{ma}_{\hat{\mathcal{R}}}(\omega^{[-i]}))$ would block some executions unnecessarily. A different choice of predicate for $p_{\hat{\mathcal{R}}}(\cdot)$ might be less conservative, but there is no guarantee that such a predicate exists.

As a concrete example, consider a security policy to require that an invocation of ι_1 be immediately followed by an invocation of ι_2. Suppose invocations of ι_1 cause monitored accesses but invocations of ι_2 do not. A reference monitor could enforce the desired security policy conservatively by blocking all executions that attempt to invoke ι_1. But that reference monitor would block some executions unnecessarily. To be less conservative, a reference monitor that receives control at an ι_1 invocation would need to use a predicate for $p_{\hat{\mathcal{R}}}(\cdot)$ that predicts whether an execution of ι_2 will immediately follow. Such a predicate

cannot exist, however, if either outcome is possible from the system state that exists when ι_1 is invoked.

Limitation: Blocking as a Mitigation. When an EM reference monitor blocks execution of S, it is to forestall an imminent violation of a security policy that is being enforced. Therefore, a security policy is not EM-enforceable if it proscribes violations that cannot be prevented by blocking further execution of S. Consider, for example, a security policy $P_T(\cdot)$ that requires S to undertake a given state transition within T seconds of some specified event. If that event has occurred but S has not yet performed the required state transition then blocking S can only make matters worse, since there is no way that an EM reference monitor can stop the advance of time. So an EM reference monitor cannot be used to enforce $P_T(\cdot)$. Other activities that continue despite blocking the execution of S include remote program executions and any external physical processes that S is controlling. Generalizing, we conclude that an EM reference monitor $\hat{\mathcal{R}}$ cannot enforce a security policy that proscribes events $\hat{\mathcal{R}}$ cannot prevent.

Security policy $P_T(\cdot)$ also exposes an implicit assumption in formal definition (11.6) of $Term_{\hat{\mathcal{R}}}(\cdot)$, which causes $P_T(Enf_{\hat{\mathcal{R}}}(\Sigma))$ to hold even though $P_T(\cdot)$ is not EM-enforceable. In formalization (11.6), τ is a finite prefix because the state remains unchanged after $\hat{\mathcal{R}}$ blocks S. That is not accurate if other on-going activities (such as the advance of time) continue changing the state (because time is part of the state) after $\hat{\mathcal{R}}$ blocks S. The finite sequences in $Term_{\hat{\mathcal{R}}}(\Sigma)$ ought to be extended by appending further state transitions caused by activities that blocking S does not stop. If we make that change to $Term_{\hat{\mathcal{R}}}(\cdot)$ then $P_T(Enf_{\hat{\mathcal{R}}}(\Sigma))$ will no longer hold, correctly indicating that $P_T(\cdot)$ would not be enforced by blocking S.

Notes and Reading

The first mention of the term "reference monitor" appears in the final report [1] of a committee[6] that the U.S. Air Force (USAF) convened to study how classified information in a computer system could be kept secure against attacks by malicious users. That report gives the three requirements a reference validation mechanism must satisfy to implement a reference monitor: tamperproof, always invoked, and small enough for assurance. The term "complete mediation" used in this chapter and elsewhere for the second requirement was introduced later in Saltzer and Schroeder [8].

In an oral history interview [9, page 63], Roger Schell, who was project manager for the USAF study, takes credit for suggesting the name "reference monitor". Schell explains in the interview that his extensive discussions with

[6]The members of the committee were: E.L. Glaser (chair), J.P. Anderson (deputy chair), Melvin Conway, Daniel J. Edwards, Hilda Faust, Steven Lipner, Eldred Nelson, Bruce Peters, Charles Rose, and Clark Weissman. Anderson's is the only name on the report cover—he drafted the report—so the report is today known as the Anderson Report.

Anderson refined the concept, which Schell admits was a version of the mediation mechanism implicit in Lampson's work [5] on the access matrix model. Schell was not the only researcher at that time exploring such mediation mechanisms. Some months before the Anderson report appeared, Graham and Denning published a paper [4] where a similar term "monitor" describes an operating system component that would be invoked with each access to objects of a given class. Denning reflects on this work in his oral history interview [2, page 38]:

> It seemed to me that we were just recording common wisdom. Maybe we were the first to put a name on it. Process managers, virtual memory managers, and file managers in operating systems all worked this way. Lampson assumed that the system implementing an access matrix worked this way. Graham and I did not think of reference monitor as the main contribution of the paper; at the time, we thought we had a good solution to the problem of mutually suspicious subsystems interacting with each other.

Although reference monitors were originally developed for mediating access to objects, the abstract idea is quite general: use state predicates to mediate state transitions attempted by monitored accesses. Schneider [10] defined EM-enforceability in order to characterize that class of security policies, introducing *security automata* as models for reference monitors. The original goal of Schneider [10] was to understand what security policies could not be enforced with some form of reference monitor, but the paper also prompted people to think about using reference monitors for controlling arbitrary state transitions. The analysis of §11.3 extends Schneider's characterization [10] by deriving limitations that arise if only certain system state transitions can be monitored accesses.

Subsequent research has extended the security automata model and the EM-enforceability results in Schneider [10]. Space does not permit a complete survey, but key contributions include the following. Viswanathan and Kim [11] discuss the computability requirements for reference monitors. To characterize security policies that can be enforced using reference monitors that perform remedial actions, Ligatti et al. [6] introduce *edit automata*, which Ligatti and Reddy [7] later generalize to obtain the more realistic family of models that they call *mandatory results automata*. Fong's [3] *shallow history automata* initiated investigations into classes of security policies that could be enforced by reference monitors where the monitor state is limited in size.

Bibliography

[1] James P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, Electronic Systems Division (AFSC), Hanscom Field, Bedford, MA, October 1972.

[2] Peter J Denning. Oral history interview with Peter J. Denning. Charles Babbage Institute. Retrieved from the University of Minnesota Digital Conservancy, April 2013.

[3] Philip W. L. Fong. Access control by tracking shallow execution history. In *2004 IEEE Symposium on Security and Privacy*, pages 43–55. IEEE Computer Society, May 2004.

[4] Scott G. Graham and Peter J. Denning. Protection: Principles and practice. In *Proceedings of the Spring Joint Computer Conference*, AFIPS '72 (Spring), pages 417–429, New York, NY, USA, May 1972. ACM.

[5] Butler W. Lampson. Protection. In *Proceedings 5th Princeton Conference on Information Sciences and Systems*, page 437, 1971. Reprinted in *ACM Operating Systems Review 8*, 1 (January 1974), page 18.

[6] Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: Enforcement mechanisms for run-time security policies. *Internatonal Journal of Information Security*, 4(1–2):2–16, February 2005.

[7] Jay Ligatti and Srikar Reddy. A theory of runtime enforcement, with results. In Dimitris Gritzali, Bart Preneel, and Marianthi Theoharidou, editors, *Computer Security – ESORICS 2010*, volume 6345 of *Lecture Notes in Computer Science*, pages 87–100, Berlin, Heidelberg, September 2010. Springer-Verlag.

[8] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, March 1975.

[9] Roger R Schell. Oral history interview with Roger R. Schell. Charles Babbage Institute. Retrieved from the University of Minnesota Digital Conservancy, May 2012.

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

[11] Mahesh Viswanathan and Moonzoo Kim. Foundations for the run-time monitoring of reactive systems – Fundamentals of the MaC language. In Zhiming Liu and Keijiro Araki, editors, *Theoretical Aspects of Computing – ICTAC 2004*, pages 543–556, Berlin, Heidelberg, 2005. Springer.

