A Theory of Runtime Enforcement, with Results

Technical Report USF-CSE-SS-102809
October 2009; Revised June 2010

Jay Ligatti Srikar Reddy

University of South Florida
Department of Computer Science and Engineering
{ligatti,sreddy4}@cse.usf.edu

Abstract. This paper presents a theory of runtime enforcement based on mechanism models called MRAs (Mandatory Results Automata). MRAs can monitor and transform security-relevant actions and their results. Because previous work could not model monitors transforming results, MRAs capture realistic behaviors outside the scope of previous models. MRAs also have a simple but realistic operational semantics that makes it straightforward to define concrete MRAs. Moreover, the definitions of policies and enforcement with MRAs are significantly simpler and more expressive than those of previous models. Putting all these features together, we argue that MRAs make good general models of runtime mechanisms, upon which a theory of runtime enforcement can be based. We develop some enforceability theory by characterizing the policies MRAs can and cannot enforce.

Key words: Security models, enforceability theory

1 Introduction

Runtime enforcement mechanisms work by monitoring untrusted applications, to ensure that those applications obey desired policies. Runtime mechanisms, which are often called runtime/security/program *monitors*, are quite popular and can be seen in operating systems, web browsers, spam filters, intrusion-detection systems, firewalls, access-control systems, stack inspection, etc. Despite their popularity and some initial efforts at modeling monitors formally, we lack satisfactory models of monitors in general, which prevents us from developing an accurate and effective theory of runtime enforcement.

1.1 Related Work

It has been difficult to model runtime mechanisms generally. Most models (e.g., [15, 17, 11, 9, 8, 1, 5]) are based on *truncation automata* [15, 12],

which can only respond to policy violations by immediately halting the application being monitored (i.e., the *target* application). This constraint simplifies analyses but sacrifices generality. For example, real runtime mechanisms often enforce policies that require the mechanisms to perform "remedial" actions, like popping up a window to confirm dangerous events with the user before they occur (to confirm a web-browser connection with a third-party site, to warn the user before downloading executable email attachments, etc). Although real mechanisms can perform these remedial actions, models based on truncation automata cannot—at the point where the target attempts to perform a dangerous action, truncation automata must immediately halt the target.

To address the limitations of truncation automata, in earlier work we proposed edit automata, models of monitors that can respond to dangerous actions by quietly suppressing them or by inserting other actions [12]. By inserting and suppressing actions, edit automata capture the practical ability of runtime mechanisms to *transform* invalid executions into valid executions, rather than the ability of truncation automata to only *recognize* and halt invalid executions. Edit automata have served as the basis for additional studies of runtime enforcement (e.g., [18, 16, 4]).

Unfortunately, while truncation automata are too limited to serve as general models of runtime mechanisms, edit automata are too powerful. The edit-automata model assumes monitors can predetermine the results of all actions without executing them, which enables edit automata to safely suppress any action. However, this assumption that monitors can predetermine the result of any action is impractical because the results of many actions are uncomputable, nondeterministic, and/or cannot tractably be predicted by a monitor (e.g., actions that return data in a network buffer, the cloud cover as read by a weather sensor, or spontaneous user input). Put another way, the edit-automata model assumes monitors can buffer—without executing—an unbounded number of target-application actions, but such buffering is impractical because applications typically require results for actions before producing new actions. For example, the echo program `x=input(); output(x)` cannot produce its second action until receiving a result, which is unpredictable, for the first. Because the echo program invokes an action that edit automata cannot suppress (due to its result being unpredictable), this simple program, and any others whose actions may not return predictable results, are outside the edit-automata model.

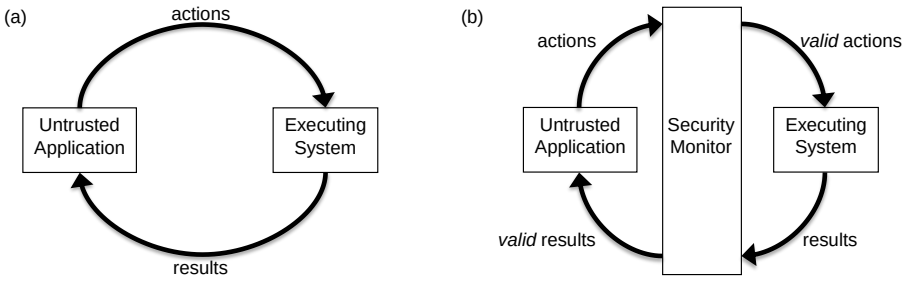

Fig. 1. In (a), an untrusted application executes actions on a system and receives results for those actions. In (b), a security monitor interposes on, and enforces the validity of, the actions executed and the results returned.

1.2 Contributions

This paper presents a theory of runtime enforcement based on mechanism models called MRAs (Mandatory Results Automata). Their name alludes to the requirement that, unlike edit automata, MRAs are obligated to return a result to the target application before seeing the next action it wishes to execute. In the MRA model, results of actions may or may not be predeterminable.

Conceptually, we wish to secure a system organized as in Figure 1a, with an application producing actions, and for every action produced, the underlying executing system (e.g., an operating system, virtual machine, or CPU) returning a result to the target application. Results may be exceptions or void or unit values, so all actions can be considered to produce results. For simplicity, this paper assumes all actions are synchronous; after the application produces an action a, it cannot produce another action until receiving a result for a. In contrast, the edit-automata model can be viewed as one in which all actions are fully asynchronous (because edit automata can buffer, without executing, an unbounded number of actions).

Figure 1b shows how we think of a monitor securing the system of Figure 1a. In Figure 1b, the monitor interposes on and transforms actions and results to ensure that the actions actually executed, and the results actually returned to the application, are valid (i.e., *satisfy* the desired policy). The monitor may or may not be inlined into the target application.

The ability of MRAs to transform *results* of actions is novel among general runtime-enforcement models, as far as we are aware. Yet this ability is crucial for enforcing many security policies, such as privacy, access-

control, and information-flow policies, which may require (trusted) mechanisms to sanitize the results of actions before (untrusted) applications access those results. For example, policies may require that system files get hidden when user-level applications retrieve directory listings, that email messages flagged by spam filters do not get returned to clients, or that applications cannot infer secret data based on the results they receive. Because existing frameworks do not model monitors transforming results of actions, one cannot use existing models to specify or reason about enforcing such *result-sanitization policies*.

The semantics of MRAs enables simple and flexible definitions of policies and enforcement—significantly simpler and more flexible than those of previous work. In particular, the definition of executions presented here allows policies to make arbitrary requirements on how monitors must transform actions and results. Consequently, this paper's definition of enforcement does not need an explicit notion of *transparency*, which previous work has considered essential for enforcement [7, 9, 12]. Transparency constrains mechanisms, forcing them to permit already-valid actions to be executed. The MRA model enables policies to specify strictly more and finer-grained constraints than transparency, thus freeing the definition of enforcement from having to hardcode a transparency requirement.

After defining MRAs and the precise circumstances under which they can be said to enforce policies, this paper briefly characterizes the sets of policies MRAs can enforce soundly, completely, and precisely.

Summary of Contributions This paper develops a theory of runtime enforcement, in which monitors may transform both actions and results. It contributes:

- A simple but general model of runtime mechanisms called MRAs. MRAs appear to be the first general model of runtime mechanisms that can transform results and enforce result-sanitization policies.
- Definitions of policies and enforcement that, because they can reason about how monitors transform actions and results, are significantly simpler and more expressive than existing definitions.
- A brief analysis of the policies MRAs can enforce soundly, completely, and precisely.

An analogy: as computability theory contributes basic models of *problems*, *algorithms*, and how algorithms may operate to *solve* problems, this line of work on enforceability theory tries to contribute basic models of *policies*, *mechanisms*, and how mechanisms may operate to *enforce* policies.

2 Background Definitions and Notation

This section briefly lays out some basic definitions of, and notation for specifying, systems and traces. The definitions and notation presented here are extended versions of definitions and notation in previous work (extended to include results of actions) [2, 15, 12].

We define a system abstractly, in terms of (1) the actions it can execute to perform computation and (2) the possible results of those actions. The system's interface determines its action set; for example, if the executing system is an operating system then actions would be system calls; if the executing system is a virtual machine then actions would be virtual-machine-code instructions (e.g., bytecode, including calls to API libraries integrated with the virtual machine); and if the executing system is machine hardware then the actions would be machine-code instructions. We use the metavariable A to represent the (nonempty, possibly countably infinite) set of actions on a system and R (disjoint from A) to represent the (nonempty, possibly countably infinite) set of results. An *event* is either an action or a result, and we use E to denote the set of events on a system; $E = A \cup R$.

An *execution* or *trace* is a possibly infinite sequence of events; it is the sequence of events that occur during a run of a monitored application and executing system. Adopting a monitor-centric view of Figure 1b, executions include events related to the monitor (1) inputting an action from the target, (2) outputting an action to the executing system, (3) inputting a result from the executing system, and (4) outputting a result to the target. To be explicit about exactly how a monitor is behaving, we subscript every event in an execution with i or o to indicate whether the monitor has input or output that event. When writing executions, we separate events by semicolons. For example, an execution could be:

$$\texttt{shutdown}_i \,;\, \texttt{popupConfirm}_o \,;\, \texttt{OK}_i \,;\, \texttt{shutdown}_o$$

This execution represents the sequence of events in which an application attempts to execute a shutdown action (so that action gets input to the monitor), to which the monitor responds by outputting a window-popup action that, when executed (e.g., by an operating system), confirms the shutdown with the user. The user OKs the shutdown, so an `OK` result gets input to the monitor, allowing the monitor to then output the shutdown action after all. This example illustrates the alternating input-output nature of monitors that arises from their role as event transformers [12].

The set of all well-formed, finite-length executions on a system with event set E is E^*; the set of all well-formed, infinite-length executions

is E^ω; and $E^\infty = E^* \cup E^\omega$. The special symbol $\cdot$ refers to the empty execution, that is, an execution in which no events occur. In general, we use $\cdot$ to refer to an absence of events; at times we use $\cdot$ to denote the absence of a single action or result. The metavariable e ranges over events, a over actions, r over results, x over executions, and $\mathcal{X}$ over sets of executions (i.e., subsets of E^∞). Sometimes it will also be convenient to use α to refer to a "potential action", that is, either $\cdot$ or an action. Similarly, ρ ranges over $\{\cdot\} \cup R$. The notation $x; x'$ denotes concatenation of two executions x and x', the result of which must be a well-formed execution (in E^∞). Finally, when x is a finite prefix of x' we write $x \preceq x'$.

3 Mandatory Results Automata

We model monitors that behave as in Figure 1b as MRAs.

3.1 Definition of MRAs

An MRA M is a tuple (E, Q, q_0, δ), where E is the event set over which M operates, Q is the finite or countably infinite set of possible states of M, q_0 is M's initial state, and δ is a (deterministic) transition function of the form $\delta : Q \times E \to Q \times E$, which takes M's current state and an event being input to M (either an action the target is attempting to execute or a result the underlying system has produced) and returns a new state for M and an event to be output from M (either an action to be executed on the underlying system or a result to be returned to the target). In contrast to earlier work [12], we do not require δ to be decidable; δ may not halt on some inputs. This ability of MRAs to diverge accurately models the abilities of real runtime mechanisms.

We call $\overset{\alpha_i}{\underset{\rho_o}{\Big|}} q \Big|^{\alpha_o}_{\rho_i}$ a *configuration* of MRA M, where q is M's current state, α_i is either $\cdot$ or the action being input to M (by the target program), α_o is either $\cdot$ or the action being output by M (to the executing system), ρ_i is either $\cdot$ or the result being input to M (by the executing system), and ρ_o is either $\cdot$ or the result being output by M (to the target program). Because MRAs process events one at a time, at most one of α_i, α_o, ρ_i, and ρ_o will ever be nonempty. Our notation for writing configurations mimics the graphic representation of monitors' inputs and outputs in Figure 1b.

We do not bother writing dots in configurations, so $\overset{a}{\Big|} q \Big|$ is the same as $\overset{a}{\underset{\cdot}{\Big|}} q \Big|^{\cdot}_{\cdot}$. The starting configuration of an MRA is $\Big| q_0 \Big|$ because the monitor

$$\frac{next_T = a}{{}_{\rho}\left|q\right| \xrightarrow{a_i} {}^{a}\left|q\right|} \textit{ (Input-Action)} \qquad \frac{next_S = r}{\left|q\right|^{a} \xrightarrow{r_i} \left|q\right|_{r}} \textit{ (Input-Result)}$$

$$\frac{\delta(q,a) = (q',a')}{{}^{a}\left|q\right| \xrightarrow{a'_o} \left|q'\right|^{a'}} \textit{ (Output-Act-for-Act)} \qquad \frac{\delta(q,r) = (q',a)}{\left|q\right|_{r} \xrightarrow{a_o} \left|q'\right|^{a}} \textit{ (Output-Act-for-Res)}$$

$$\frac{\delta(q,a) = (q',r)}{{}^{a}\left|q\right| \xrightarrow{r_o} {}_{r}\left|q'\right|} \textit{ (Output-Res-for-Act)} \qquad \frac{\delta(q,r) = (q',r')}{\left|q\right|_{r} \xrightarrow{r'_o} {}_{r'}\left|q'\right|} \textit{ (Output-Res-for-Res)}$$

Fig. 2. Single-step semantics of mandatory results automata.

begins executing in its initial state with no events yet input or output.

We define the operational semantics of MRAs with a labeled single-step judgment whose form is $C \xrightarrow{e}_M C'$. This judgment indicates that MRA M takes a single step from configuration C to configuration C' while extending the current trace by event e (which will be tagged as either an input or output event). Because M will always be clear from context, we henceforth omit it from the judgment.

The definition of MRAs' single-step semantics appears in Figure 2. Six inference rules define all possible MRA transitions:

1. *Input-Action* enables the MRA to receive a new input action from the target ($next_T$ is the next action generated by the target). Because ρ ranges over $\{\cdot\} \cup R$, the MRA can receive a new input action when in its initial configuration $\left|q_0\right|$ or when in a configuration of the form ${}_{r}\left|q\right|$ (in which case the MRA has most recently returned a result r to the target, so it is ready for another input action).
2. *Output-Act-for-Act* enables the MRA, immediately after inputting action a, to output a possibly different action a'.
3. *Output-Res-for-Act* enables the MRA, immediately after inputting action a, to return a result r for a to the target.
4. *Input-Result* enables the MRA to receive a new input result r for its most recent output action a ($next_S$ is the next result generated by the system).
5. *Output-Act-for-Res* enables the MRA, immediately after inputting result r, to output another action a.

6. *Output-Res-for-Res* enables the MRA, immediately after inputting result r, to return a possibly different result r' to the target for the action it most recently tried to execute.

Although many alternatives exist for defining MRAs' semantics (e.g., process calculi and other deductive systems, some of which can be compressed into four inference rules), we carefully selected the rules in Figure 2 based on their simplicity—not just in the rules themselves but also in the transition functions of MRAs that step according to those rules.

Several observations about the operational semantics:

- MRAs can "accept" an input action a by outputting it (with *Output-Act-for-Act*), receiving a result r for a (with *Input-Result*), and then returning r to the application (with *Output-Res-for-Res*).
- MRAs can "halt" an application by outputting an action like `exit`, if the underlying system can execute such an action, or by entering an infinite loop in its transition function to block additional inputs and outputs from being made.
- MRAs are indeed obligated to return results to applications before inputting new actions. No transitions allow an MRA to input another action until it has discharged the last by returning a result for it.
- MRAs can avoid or postpone executing dangerous actions while allowing the target to continue executing (with *Output-Res-for-Act*). For example, an MRA could avoid executing a dangerous port-open input action by outputting an error-code or exception result in response. Alternatively, the MRA could quietly postpone executing the port-open action by immediately outputting a `void` result and then observing how the target uses the port; if the target uses the port securely then the MRA could output the original port-open action followed by the secure port-use action(s) (with *Output-Act-for-Act* and *Output-Act-for-Res*). By postponing (i.e., buffering) dangerous actions until they are known to be secure, MRAs can operate as edit automata; however, such buffering is only possible when the valid results of buffered actions are predictable (such as `void` results of some port-open actions).
- We make no assumptions about whether and how the executing system generates results for actions; the executing system may produce results nondeterministically or through uncomputable means (e.g., by reading a weather sensor or spontaneous keyboard input). This design captures the reality that monitors can only determine the results of many actions (e.g., `readFile`, or `getUserInput`) by having the system actually execute those actions. Hence, the *Input-Result* transition,

and the single-step relation for MRAs in general, may be nondeterministic. Similarly, MRAs have no knowledge of whether and how the target generates actions, so the *Input-Action* transition may be nondeterministic as well.

These observations, and the semantics of MRAs in general, match our understanding of how real program monitors behave. For example, in the Polymer enforcement system [3], policies can make *insertion suggestions* to output arbitrary actions in response to an input action, can make *exception* or *replacement suggestions* to output an arbitrary result for the most recent input action, can monitor results of actions, and must return a result for the most recent input action before inputting another. PSLang and LoPSiL policies, and aspects in many languages (e.g., AspectJ), behave similarly [7, 13, 10].

Limitations Nonetheless, because MRAs are models, some gaps do exist between the possible behaviors of MRAs and what real monitors can do in practice. MRAs share two standard limitations with other general runtime-enforcement models: (1) MRAs can interpose on and make decisions about *all* security-relevant actions and results, but in practice some events may be imperceptible to the monitor (e.g., monitoring every "clock-tick" action is possible in our model but impractical); this is a problem of *complete mediation* [14], and (2) by executing transition functions, MRAs may delay the processing of time-sensitive events, which prevents MRAs from enforcing some time-sensitive policies (this issue is inherent in runtime monitoring). Besides these standard limitations, MRAs have another: for simplicity in this paper, MRAs treat all actions as synchronous (i.e., they finish processing, and return a result for, one input action before inputting another). This limitation prevents MRAs from effectively monitoring applications whose correctness depends on some security-relevant action(s) being asynchronous. However, as mentioned in Section 1.2, the edit-automata model already provides a semantics for monitoring asynchronous actions.

3.2 Example MRAs

We next consider a couple example MRAs exhibiting simple, everyday sorts of behaviors found in practical monitors. The behaviors are so simple that they may seem trivial; nonetheless, the behaviors are outside existing runtime-enforcement models because they involve monitors acting on unpredictable results of actions (something neither truncation nor edit automata can do).

Example 1: Spam-filtering MRA This MRA M sanitizes the results of `getMessages` actions to filter out spam emails. M's state consists of a boolean flag indicating whether M is in the process of obtaining email messages; M begins in state 0. M's transition function δ is:

$$\delta(q,e) = \begin{cases} (0,e) & \text{if } q = 0 \text{ and } e \neq \texttt{getMessages} \\ (1,e) & \text{if } q = 0 \text{ and } e = \texttt{getMessages} \\ (0, filter(e)) & \text{if } q = 1 \end{cases}$$

That is, M outputs its inputs verbatim and does not change its state as long as it does not input a `getMessages` action. When M does input `getMessages`, it sets its boolean flag and allows `getMessages` to execute. If M then inputs a result r for `getMessages`, it outputs the spam-filtered version of r and returns to its initial state. With similar techniques, M could sanitize results in other ways (e.g., to remove system files from directory listings).

Example 2: Dangerous-action-confirming MRA Our second example MRA pops up a window to confirm a dangerous action d with the user before allowing d to execute. We assume d has a default return value r, which must be returned when the user decides not to allow d to execute (r would typically be a `null` pointer or a value indicating an exception). We also assume a `popupConfirm` action that works like a `JOptionPane.showConfirmDialog` method in Java, returning either an `OK` or `cancel` result. M uses a boolean flag, again initially set to 0, for its state, and the following transition function.

$$\delta(q,e) = \begin{cases} (0,e) & \text{if } q = 0 \text{ and } e \neq d \\ (1, \texttt{popupConfirm}) & \text{if } q = 0 \text{ and } e = d \\ (0,r) & \text{if } q = 1 \text{ and } e = \texttt{cancel} \\ (0,d) & \text{if } q = 1 \text{ and } e = \texttt{OK} \end{cases}$$

This function works as expected: M outputs non-d input events verbatim. Once M inputs a d action, it outputs a `popupConfirm` action and waits for a result. If the user cancels the execution of d, M outputs result r; otherwise it outputs action d.

Summary: Because of the simplicity in MRAs' operational semantics, and in concrete MRA transition functions, plus the fact that MRA behaviors match our understanding of the essential behaviors of real runtime monitors, we believe MRAs serve as a good basis for developing a theory of runtime enforcement.

3.3 Generalizing the Operational Semantics

Before we can formally define what it means for an MRA to enforce a policy, we need to generalize the single-step semantics to account for multiple steps. First, we define the (finite) multi-step relation, with judgment form $C \xrightarrow{x}{}^{*} C'$, in the standard way as the reflexive, transitive closure of the single-step relation. The trace above the arrow in the multi-step judgment gets built by concatenating, in order, every event labeled in the single-step judgments. Hence, $C \xrightarrow{x}{}^{*} C'$ means that the MRA builds execution x while transitioning, using any finite number of single steps, from configuration C to configuration C'.

We also define a judgment $M \Downarrow x$ to mean that MRA M, when its input events match the sequence of input events in x, in total *produces* the possibly infinite-length trace x. To define $M \Downarrow x$ formally, there are two cases to consider: First, when $x \in E^{\omega}$, $M \Downarrow x$ iff for all prefixes x' of x, there exists an M-configuration C such that $C_0 \xrightarrow{x'}{}^{*} C$ (where C_0 is M's initial configuration). Second, when $x \in E^{*}$, $M \Downarrow x$ iff there exists an M-configuration C such that (1) $C_0 \xrightarrow{x}{}^{*} C$ and (2) if x ends with an input event then M never transitions from C (otherwise, x would not be the *entire* trace produced on x's input events).

4 MRA-based Enforcement

This section defines what it means for an MRA to enforce a policy.

4.1 Policies and Properties

A *policy* is a predicate on sets of executions [15]; a set of executions $\mathcal{X} \subseteq E^{\infty}$ satisfies policy P iff $P(\mathcal{X})$. Some policies are also *properties*. Policy P is a property iff there exists a predicate $\hat{P}$ over E^{∞} such that $\forall \mathcal{X} \subseteq E^{\infty} : \Big(P(\mathcal{X}) \iff \forall x \in \mathcal{X} : \hat{P}(x)\Big)$. There is a one-to-one correspondence between a property P and its predicate $\hat{P}$, so the rest of the paper uses $\hat{P}$ unambiguously to refer to both.

Intuitively, policies can determine whether a set of target executions is valid based on the executions' relationships with one another, but properties cannot take such inter-execution relationships into account. It is sometimes possible for runtime mechanisms to enforce nonproperty policies: a monitor could refer to earlier traces (e.g., saved in files) when deciding how to transform the current execution, or it could monitor multiple executions of a program concurrently [6]. For simplicity, though,

this paper analyzes only the properties MRAs can enforce; we assume monitors make decisions about a single execution at a time.

There are two important differences between this paper's definition of policies and the definitions in previous models. The differences arise from the way executions are modeled here: instead of modeling executions as just the actions a monitor outputs, the MRA model also includes (1) output results, and (2) all input events, in executions. Because policies here may take output results into account, they can specify constraints on which results may be returned to targets; policies here may require results to be sanitized. For example, the spam-filtering MRA from Section 3.2 enforces a policy requiring all results of `getMessages` actions to be filtered (this policy is a property because it is satisfied iff *every* execution in a set $\mathcal{X}$ has exactly zero spam-containing results of `getMessages` actions).

Moreover, because policies in the MRA model can take input events into account, **policies here can require *arbitrary* relationships to hold between input and output events**. For example, a property $\hat{P}$ could be dissatisfied by execution $\texttt{shutdown}_i$ (i.e., $\neg\hat{P}(\texttt{shutdown}_i)$) but be satisfied by $\texttt{shutdown}_i\,;\texttt{popupConfirm}_o$. To enforce this $\hat{P}$, an MRA may have no choice but to output `popupConfirm` upon inputting a `shutdown` action. Policies in previous models (e.g., truncation and edit automata) could not specify such relationships between input and output events because the policies were predicates over output executions only. The only relationship allowed between input and output events in previous models was *transparency*, which was hardcoded into the definition of enforcement [9, 12] and required monitors to output valid inputs unchanged. Transparency can be encoded in policies in the MRA model (by defining policies to be satisfied only by executions in which valid inputs get output unchanged), but **policies here are strictly more expressive than transparency** because they can specify arbitrary input-output relationships. For example, the popup-confirmation policy above specifies a relationship that is outside the scope of transparency (because there is no requirement for `shutdown` to be output unchanged).

4.2 Enforcement

We define enforcement in terms of standard principles of soundness and completeness. MRA M is *sound* with respect to property $\hat{P}$ when M only produces traces satisfying $\hat{P}$; M is *complete* with respect to $\hat{P}$ when it produces all traces satisfying $\hat{P}$; and M is *precise* with respect to $\hat{P}$ when it is sound and complete with respect to $\hat{P}$.

Definition 1. *On a system with event set E, MRA M:*

- **soundly enforces** $\hat{P}$ *iff* $\forall x \in E^\infty : ((M\Downarrow x) \implies \hat{P}(x))$,
- **completely enforces** $\hat{P}$ *iff* $\forall x \in E^\infty : (\hat{P}(x) \implies (M\Downarrow x))$, *and*
- **precisely enforces** $\hat{P}$ *iff* $\forall x \in E^\infty : ((M\Downarrow x) \iff \hat{P}(x))$.

Definition 1 is significantly simpler and more flexible than definitions of enforcement in related work, because it (1) does not hardcode transparency requirements, and (2) defines complete and precise, in addition to sound, enforcement.

For an example of MRA enforcement, we reconsider the dangerous-action-confirming MRA M of Section 3.2 (recall that M pops up a window to get user confirmation before executing action d; if the user cancels execution of d, a result r gets returned to the target in place of executing d). Let us use $e_i; e_o$ as shorthand for any two-event sequence in which a non-d event is input and then immediately output. Then M precisely enforces a property $\hat{P}$ satisfied by exactly those well-formed executions matching the pattern:

$$(e_i; e_o \mid d_i; \texttt{popupConfirm}_o(\texttt{cancel}_i; r_o \mid \texttt{OK}_i; d_o))^\infty \ (d_i; \texttt{popupConfirm}_o)?$$

This pattern exactly characterizes the executions M builds. M outputs its input events verbatim until no additional inputs arrive or a d action is input. Once M inputs a d action, it immediately outputs the `popupConfirm` action. Execution stops at this point if the user never selects whether d should be allowed; the pattern above therefore allows executions to optionally end with $d_i; \texttt{popupConfirm}_o$. However, if M inputs a `cancel` or `OK` result for the `popupConfirm` action, it must output the appropriate event in response (either r or d) and continue executing. Note that this policy disallows executions from just ending with, for example, an `OK` result being input to confirm a d action; the policy requires execution to continue after the `OK` input by allowing d to execute. The policy therefore specifies a non-transparency relationship between input and output events (unrelated to outputting inputs unchanged), which cannot be expressed in previous enforcement models.

4.3 Wanted: Auxiliary Predicates, Dead and Alive

Given a property $\hat{P}$ and a finite execution x, we often find it useful to know whether x can be extended into a valid execution. We introduce two predicates for this purpose: when x can be made valid by extending it with

some sequence of events, we say x is *alive*; otherwise, x is *dead*. Formally, $alive_{\hat{P}}(x) \iff (\exists x' \in E^\infty : \hat{P}(x; x'))$ and $dead_{\hat{P}}(x) \iff \neg alive_{\hat{P}}(x)$.

Because $\hat{P}$ will always be clear from context, we omit it as a subscript in future $alive(x)$ and $dead(x)$ judgments. Also, because properties in practice generally have predicates $\hat{P}$ and *alive* that are decidable over finite-length inputs, and because only considering such properties simplifies the theorems in Section 5, this paper limits its scope to properties with predicates $\hat{P}(x)$ and $alive(x)$ that are decidable over finite x.

5 Analysis of MRA-enforceable Policies

This section characterizes the properties MRAs can soundly, completely, and precisely enforce. For each of these types of enforcement, we define the set of properties MRAs can enforce in that way.

5.1 Sound Enforcement

The primary limitation of MRAs to soundly enforce a property arises from their inability to predict, after outputting an event, whether another input event will arrive. Consequently, sound MRAs must ensure that their produced executions are always valid after outputting an event.

Theorem 1 formalizes the properties MRAs can soundly enforce[1]. Soundly enforceable properties have three traits. First, there must be a recursively enumerable predicate R over E^* that separates the executions an MRA can produce from those it cannot; $R(x) \iff M \Downarrow x$. Then we have $R(\cdot)$ because $M \Downarrow \cdot$ (for all M). Second, any time an MRA receives an input event e after having already produced execution x (i.e., $R(x)$), either the MRA can soundly cease producing an execution (by entering an infinite loop) because $\hat{P}(x; e_i)$, or there must exist some valid way for the MRA to respond to e (by outputting some valid event e' to produce $x; e_i; e'_o$). Finally, to make sure an MRA never produces an invalid infinite-length execution x, there must be some unproducible prefix of x that ends with an output event.

[1] Notes on notation in this section: We write $\exists (x'; e_i) \preceq x : F$ to mean that there exists a prefix of x having the form $x'; e_i$ such that F holds. Similarly, the notation $\forall (x; e_i) \in E^* : F$ means that F is true for all well-formed finite executions $x; e_i$. We also use uniqueness quantification of the form $\exists_1 e \in E : F$ to mean that there exists exactly one event e in E such that F is true. Finally, we assume conjunction ($\wedge$) binds more tightly than disjunction ($\vee$).

Theorem 1. *Property $\hat{P}$ on a system with event set E can be* **soundly** *enforced by some MRA M iff there exists recursively enumerable predicate R over E^* such that all the following are true.*

1. $R(\cdot)$
2. $\forall (x; e_i) \in E^* : \left(\neg R(x) \ \vee \ \hat{P}(x; e_i) \ \vee \ \exists e' \in E : \begin{pmatrix} R(x; e_i; e'_o) \\ \wedge \ \hat{P}(x; e_i; e'_o) \end{pmatrix} \right)$
3. $\forall x \in E^\omega : \left(\neg \hat{P}(x) \implies \exists (x'; e_i) \preceq x : \neg R(x') \right)$

Proofs are in Appendix A.

5.2 Complete Enforcement

The necessary-and-sufficient conditions for completely enforcing a property are a bit simpler than those for sound enforcement. When considering any finite execution ending in an input event (i.e., execution $x; e_i$), two possibilities exist. First, if $x; e_i$ is valid, a complete-enforcer MRA has to produce exactly $x; e_i$, but doing so prevents it from outputting an event in response to e (outputting an e' in response to e would make the MRA produce $x; e_i; e'_o$, not $x; e_i$). Hence, $dead(x; e_i; e'_o)$ for all $e' \in E$ in this case. On the other hand, if $x; e_i$ is not valid, the MRA either (1) never produces x, in which case it can't produce any extensions of x, so they all must be dead, or (2) if the MRA can produce x then after doing so, on input e there must be at most one possible e' that the MRA can find and output to lead to a valid trace—if there were multiple outputs leading to valid traces, the MRA could not completely produce all valid traces because it can only output at most one event for every event input. Theorem 2 formalizes these ideas.

Theorem 2. *Property $\hat{P}$ on a system with event set E can be* **completely** *enforced by some MRA M iff:*

$$\forall (x; e_i) \in E^* : \begin{pmatrix} \forall e' \in E : dead(x; e_i; e'_o) \\ \vee \ \neg \hat{P}(x; e_i) \ \wedge \ \exists_1 e' \in E : alive(x; e_i; e'_o) \end{pmatrix}$$

5.3 Precise Enforcement

To learn which policies MRAs precisely enforce, we can intersect the policies soundly enforceable with the policies completely enforceable, and then simplify the result. Doing so produces Theorem 3.

Theorem 3. *Property $\hat{P}$ on a system with event set E can be* **precisely** *enforced by some MRA M iff all the following are true.*

1. $\hat{P}(\cdot)$
2. $\forall (x; e_i) \in E^* : \left(\begin{array}{l} \neg \hat{P}(x) \\ \vee\ \hat{P}(x; e_i)\ \wedge\ \forall e' \in E : dead(x; e_i; e'_o) \\ \vee\ \neg \hat{P}(x; e_i)\ \wedge\ \exists_1 e' \in E : \hat{P}(x; e_i; e'_o) \\ \qquad\qquad\qquad \wedge\ \exists_1 e' \in E : alive(x; e_i; e'_o) \end{array} \right)$
3. $\forall x \in E^\omega : \left(\neg \hat{P}(x) \implies \exists (x'; e_i) \preceq x : \neg \hat{P}(x') \right)$

Intuitively, Theorem 3 says that MRAs *only* precisely enforce properties satisfying three constraints. First, the empty execution has to be valid because every MRA produces $\cdot$ when no inputs arrive. Second, when considering any execution $x; e_i$ such that the precise-enforcer M produces x, there are two possibilities: (1) after producing x, M has to produce $x; e_i$ because it's valid, but doing so prevents M from producing any other valid extensions of $x; e_i$, or (2) after producing x and inputting an invalid e, M has to find and output the one and only event e' that makes its produced trace valid (for soundness) and alive (for completeness). Finally, because M cannot produce any invalid infinite-length execution x, there must always be some invalid (i.e., unproducible) prefix of such an x.

Conversely, to construct an MRA M to precisely enforce any property $\hat{P}$ satisfying the constraints of Theorem 3, let M track the execution being produced. When M has produced x and then inputs e, simply have M search for, and output if found, an $e' \in E$ such that $\hat{P}(x; e_i; e'_o)$ (the theorem's second constraint ensures that at most one such e' exists). Defined in this way, M only produces valid traces (because initially $\cdot$ is valid, and M maintains the validity of its produced traces) and produces all valid traces (because it only gives up on execution paths when they're known to be dead).

6 Conclusions

We have presented MRAs as general models of runtime enforcement mechanisms. MRAs do not suffer from the primary problems of previous models because they (1) allow monitors to transform actions and results and (2) do not assume that monitors can predetermine results of actions. MRAs are the first general models of runtime enforcement we know of in which result-sanitization policies can be reasoned about, and we have seen some examples of how MRAs with simple transition functions can enforce result-sanitization, and other result-dependent, policies. Also, the definitions of policies and enforcement with MRAs are significantly simpler and

more expressive than existing definitions because they allow policies to require arbitrary (including non-transparency) relationships between input and output events. Finally, after defining MRAs and enforcement, we have characterized the policies they soundly, completely, and precisely enforce, so for example, security engineers should never waste effort attempting to use MRA-style mechanisms to precisely enforce policies outside the set defined by Theorem 3.

These contributions, and theories of runtime enforcement in general, are important because they:

- shape how we think about the roles and meanings of policies, mechanisms, and enforcement,
- influence our decisions about how to specify policies and mechanisms (including designs of policy-specification languages),
- enable us to reason about whether specific mechanisms enforce desired policies, and
- improve our understanding of which policies can and cannot be enforced at runtime.

We hope that with continued research, enforceability theory will benefit the security community in similar ways that computability theory has benefited the broader computer-science community.

Acknowledgments

We are grateful for feedback from Lujo Bauer, Egor Dolzhenko, Frank Piessens, and the anonymous reviewers. This research was supported by National Science Foundation grants CNS-0716343 and CNS-0742736.

References

1. I. Aktug, M. Dam, and D. Gurov. Provably correct runtime monitoring. In *Proceedings of the 15th International Symposium on Formal Methods*, May 2008.
2. B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, Oct. 1985.
3. L. Bauer, J. Ligatti, and D. Walker. Composing expressive runtime security policies. *ACM Transactions on Software Engineering and Methodology*, 18(3):1–43, 2009.
4. D. Beauquier, J. Cohen, and R. Lanotte. Security policies enforcement using finite edit automata. *Electron. Notes Theor. Comput. Sci.*, 229(3):19–35, 2009.
5. M. Dam, B. Jacobs, A. Lundblad, and F. Piessens. Security monitor inlining for multithreaded java. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, July 2009.

6. D. Devriese and F. Piessens. Non-interference through secure multi-execution. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 109–124, May 2010.
7. Ú. Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Cornell University, Jan. 2004.
8. P. W. L. Fong. Access control by tracking shallow execution history. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2004.
9. K. Hamlen, G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *ACM Transactions on Progamming Languages and Systems*, 28(1):175–205, Jan. 2006.
10. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *European Conference on Object-oriented Programming*. Springer-Verlag, 2001.
11. M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswantathan. Computational analysis of run-time monitoring—fundamentals of Java-MaC. In *Run-time Verification*, June 2002.
12. J. Ligatti, L. Bauer, and D. Walker. Run-time enforcement of nonsafety policies. *ACM Transactions on Information and System Security*, 12(3):1–41, Jan. 2009.
13. J. Ligatti, B. Rickey, and N. Saigal. LoPSiL: A location-based policy-specification language. In *International ICST Conference on Security and Privacy in Mobile Information and Communication Systems (MobiSec)*, June 2009.
14. J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, Sept. 1975.
15. F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and Systems Security*, 3(1):30–50, Feb. 2000.
16. C. Talhi, N. Tawbi, and M. Debbabi. Execution monitoring enforcement under memory-limitation constraints. *Information and Computation*, 206(2–4):158–184, 2008.
17. M. Viswanathan. *Foundations for the Run-time Analysis of Software Systems*. PhD thesis, University of Pennsylvania, 2000.
18. D. Yu, A. Chander, N. Islam, and I. Serikov. Javascript instrumentation for browser security. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 237–249, 2007.

A Proofs

Theorem 1. *Property $\hat{P}$ on a system with event set E can be* soundly *enforced by some MRA M iff there exists recursively enumerable predicate R over E^* such that all the following are true.*

1. $R(\cdot)$
2. $\forall (x; e_i) \in E^* : \left(\neg R(x) \ \vee \ \hat{P}(x; e_i) \ \vee \ \exists e' \in E : \begin{pmatrix} R(x; e_i; e'_o) \\ \wedge \ \hat{P}(x; e_i; e'_o) \end{pmatrix} \right)$
3. $\forall x \in E^\omega : \left(\neg \hat{P}(x) \implies \exists (x'; e_i) \preceq x : \neg R(x') \right)$

Proof. For the *if* direction, consider any $\hat{P}$ and R satisfying the three constraints. An MRA M can be constructed to soundly enforce such a $\hat{P}$ by having M's state keep track of the current execution produced. M will maintain the invariant that whenever it's ready to input an action, its execution satisfies R; Constraint 1 in the theorem statement guarantees this invariant holds initially. Upon inputting an event e, M checks whether its current execution (now $x; e_i$, where $R(x)$) is valid; if it is then M enters an infinite loop; otherwise, M finds and outputs some $e' \in E$ that will make the current execution satisfy both R and $\hat{P}$ (some such e' must exist by Constraint 2 in the theorem statement). M soundly enforces $\hat{P}$ because (1) if M receives a finite number of input events, it guarantees the validity of its produced trace, either by halting on a valid input or by outputting a valid event in response to its final input, and (2) if M receives an infinite number of input events, its produced trace cannot be an invalid $x \in E^\omega$ because Constraint 3 in the theorem statement would require there to exist a prefix of x just before an input event that M cannot produce but would have to if it were to go on to produce x.

For the *only-if* direction, consider any MRA M soundly enforcing a property $\hat{P}$. Let $R(x)$ hold when $M \Downarrow x$, which makes R a recursively enumerable predicate over E^*. This definition of R implies Constraint 1, that $R(\cdot)$. To prove Constraint 2, consider any execution $x; e_i \in E^*$ such that $R(x)$ and $\neg\hat{P}(x; e_i)$, meaning that M produces x and must then output some valid event e' in response to input e (because after outputting e', M may receive no additional inputs and therefore have no further opportunities to extend its trace). Hence, $R(x; e_i; e'_o)$ and $\hat{P}(x; e_i; e'_o)$. Finally, if Constraint 3 did not hold then there would exist an invalid infinite-length x whose prefixes (ending in output events) are all producible by M, implying $M \Downarrow x$ while $\neg\hat{P}(x)$, which would contradict M's soundness. □

Theorem 2. *Property $\hat{P}$ on a system with event set E can be* completely *enforced by some MRA M iff:*

$$\forall (x; e_i) \in E^* : \begin{pmatrix} \forall e' \in E : dead(x; e_i; e'_o) \\ \vee\ \neg\hat{P}(x; e_i)\ \wedge\ \exists_1 e' \in E : alive(x; e_i; e'_o) \end{pmatrix}$$

Proof. For the *if* direction, consider any $\hat{P}$ satisfying the theorem's constraint. An MRA M can be constructed to completely enforce such a $\hat{P}$ by having M's state keep track of the current execution produced. Upon inputting an event e, M checks whether its current execution (now $x; e_i$)

is valid; if it is then M enters an infinite loop; otherwise, M searches for and, if found, outputs some $e' \in E$ such that $x; e_i; e'_o$ is alive (at most one such e' may exist by the theorem's constraint). M completely enforces $\hat{P}$ because it guarantees that it produces all alive traces (every alive trace ending with an input event can be extended in at most one way to remain alive, and M always finds and extends the trace in that way if possible). Because every valid execution $x \in E^\infty$ is either alive (if $x \in E^*$) or has entirely alive prefixes (if $x \in E^\omega$), M produces every valid execution.

For the *only-if* direction, consider any MRA M completely enforcing a property $\hat{P}$ and any execution $x; e_i \in E^*$. If $\hat{P}(x; e_i)$ then $M \Downarrow x; e_i$ because M is a complete enforcer, but according to the definition of execution production, M cannot transition after producing $x; e_i$, so $\forall e' \in E : dead(x; e_i; e'_o)$. On the other hand, if $\neg\hat{P}(x; e_i)$ then there are two subcases: either M does or does not produce x. In the first subcase, after M produces x and then receives e as input, it may output at most one event in response, so for M to be a complete enforcer, there must be at most one $e' \in E$ such that $alive(x; e_i; e'_o)$. In the second subcase, M never produces x, so all extensions of x must be dead. □

Theorem 3. *Property $\hat{P}$ on a system with event set E can be* precisely *enforced by some MRA M iff all the following are true.*

1. $\hat{P}(\cdot)$
2. $\forall (x; e_i) \in E^* : \left(\begin{array}{l} \neg\hat{P}(x) \\ \vee\ \hat{P}(x; e_i)\ \wedge\ \forall e' \in E : dead(x; e_i; e'_o) \\ \vee\ \neg\hat{P}(x; e_i)\ \wedge\ \exists_1 e' \in E : \hat{P}(x; e_i; e'_o) \\ \qquad\qquad \wedge\ \exists_1 e' \in E : alive(x; e_i; e'_o) \end{array} \right)$
3. $\forall x \in E^\omega : \left(\neg\hat{P}(x) \implies \exists (x'; e_i) \preceq x : \neg\hat{P}(x') \right)$

Proof. By Definition 1 and Theorems 1 and 2. The three constraints in this theorem are obtained by intersecting the constraints in Theorems 1 and 2 and then simplifying the result based on two observations: (1) because precise enforcers are sound and complete, $\forall (x; e_i) \in E^* : (R(x) \iff \hat{P}(x))$, where R is the predicate from Theorem 1, and (2) the constraints resulting from the intersection imply that if $\neg\hat{P}(x)$ (where again $(x; e_i) \in E^*$) then $dead(x)$. □