Noname manuscript No.
(will be inserted by the editor)

Modeling Runtime Enforcement with Mandatory Results Automata

Egor Dolzhenko · Jay Ligatti · Srikar Reddy

Received: date / Accepted: date

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 general security monitors transforming results, MRAs capture realistic behaviors outside the scope of previous models. MRAs also have a simple operational semantics that makes it straightforward to define concrete MRAs. Moreover, the definitions of policies and enforcement with MRAs are 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 deterministic and nondeterministic MRAs can and cannot enforce.

Keywords Enforceability Theory · Monitoring · Runtime Enforcement · Security Automata · Models of Enforcement

Egor Dolzhenko
Department of Computer Science and Engineering, and Department of Mathematics and Statistics
University of South Florida E-mail: edolzhen@mail.usf.edu

Jay Ligatti
Department of Computer Science and Engineering
University of South Florida
E-mail: ligatti@cse.usf.edu

Srikar Reddy
Department of Computer Science and Engineering
University of South Florida
E-mail: sreddy4@cse.usf.edu

1 Introduction

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

Not having general models of runtime mechanisms is problematic because it prevents us from developing an accurate and effective theory of runtime enforcement. On the other hand, if we can model runtime mechanisms accurately and generally, we should be able to use those models to better understand how real security mechanisms operate and what their limitations are, in terms of policies they can and cannot enforce.

1.1 Related Work

It has been difficult to model runtime mechanisms generally. Most models (e.g., [22,24,16,13,12,1,8]) are based on *truncation automata* [22,18], 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 typically 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

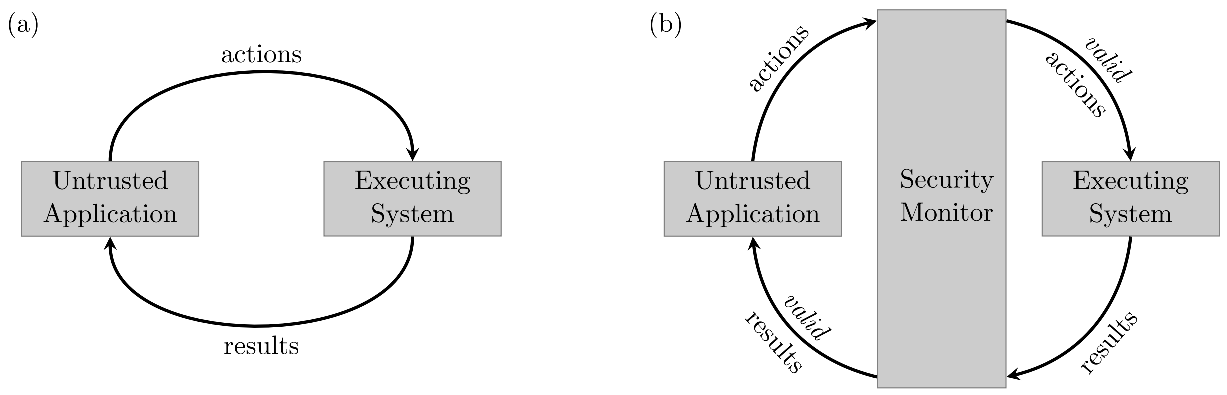

Fig. 1 In (a), an untrusted application executes actions on a system and receives results for those actions. In (b), a security monitor interposes on, and enforces the validity of, the actions executed and the results returned.

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. Immediately halting the target application without performing some auxiliary actions (e.g., popping up a window or writing to a log file) to audit or otherwise explain the program termination is not user friendly. We know of no runtime mechanisms operating as true truncation automata in practice.

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 [18]. By inserting and suppressing actions, edit automata capture the ability of practical 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., [25, 23, 5]).

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 in general because applications may 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.

More recently, Basin, Jugé, Klaedtke, and Zălinescu have considered a model in which monitors may observe but not suppress some actions [3]. These uncontrollable actions, the primary example being "clock ticks", must be output verbatim by monitors. The present paper considers the more general problem of constraining the ways in which monitors' outputs must relate to their inputs, including results of actions. Section 5 shows how the particular case of uncontrollable events can be encoded into this paper's framework.

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, where an application produces actions, and for every action produced, the underlying executing system (e.g., an operating system, virtual machine, or CPU) returns 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, the

MRA model assumes synchonrizable actions: after the monitor inputs an application action a, it cannot input another action until it has output a result for a. In practice this behavior may be implemented with locks; the monitor acquires a lock on action input and releases that lock on result output. Hence, MRAs cannot enforce policies on applications whose correctness would be violated by having their monitored actions synchronized (i.e., serialized) in this way. For many applications, even multithreaded applications, synchronization of monitored actions decreases performance but does not affect correctness (e.g., the case study in [4] used locks as described above to monitor a multithreaded email client, without affecting its correctness).

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 (though many papers have modeled monitors transforming results in particular domains, such as usage control [21,20] and policy composition [4], without investigating the enforcement capabilities of such monitors). Yet this ability to transform results 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 in general 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. 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 [10,13,18]. 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 characterizes the policies MRAs can enforce soundly, completely, and precisely. The paper then generalizes MRAs by introducing nondeterministic MRAs (NMRAs) and analyzes their enforcement powers as well. NMRAs model situations in which external factors (such as a thread scheduler) influence a monitor's behavior. Finally, we compare the analyses of MRAs and NMRAs to derive a hierarchy of policies they 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 simpler and more expressive than existing definitions.
– A generalization of MRAs to NMRAs, which model runtime mechanisms that may execute nondeterministically (e.g., because they're implemented in multiple threads or processes).
– An analysis of the policies MRAs and NMRAs can enforce soundly, completely, and precisely.
– A hierarchy of runtime-enforceable policies based on the previously mentioned analysis.

This article is based on work first presented at ESORICS 2010 [19].

2 Background Definitions and Notation

This section defines actions, results, executions, and operations on them. Table 1 in Appendix A summarizes the symbols introduced and used in this paper.

An *alphabet* is a finite set of symbols. A *word* over the alphabet Σ is a finite sequence of symbols from Σ; e.g., *abb* is a word over the alphabet $\Sigma = \{a, b\}$. The set of (nonempty) words is denoted by Σ^+.

We use a nonempty set of words $A \subset \Sigma^+$ to represent the set of *actions* a system can execute and another nonempty set of words $R \subset \Sigma^+$ to represent the set of those actions' *results*. The sets A and R are assumed to be computable and disjoint. An *event* is either an action or a result. We use E to denote the set of events; $E = A \cup R$. An *exchange* ξ is a pair of events $\langle e, e' \rangle$,

$$\boxed{C \xrightarrow{\xi} C'}$$

$$\frac{\text{next}_t = a \quad \delta(q,a) = (q',a')}{q_t \xrightarrow{\langle a,a' \rangle} q'_s}\ \textit{(In-Act-Out-Act)} \qquad \frac{\text{next}_s = r \quad \delta(q,r) = (q',a)}{q_s \xrightarrow{\langle r,a \rangle} q'_s}\ \textit{(In-Res-Out-Act)}$$

$$\frac{\text{next}_t = a \quad \delta(q,a) = (q',r)}{q_t \xrightarrow{\langle a,r \rangle} q'_t}\ \textit{(In-Act-Out-Res)} \qquad \frac{\text{next}_s = r \quad \delta(q,r) = (q',r')}{q_s \xrightarrow{\langle r,r' \rangle} q'_t}\ \textit{(In-Res-Out-Res)}$$

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

consisting of an *input event* e (i.e., an event input to the monitor) and an *output event* e' (i.e., an event output from the monitor). The set of all exchanges over E is denoted by $\mathcal{E}$. Given sets E and E' of events, define $\langle E, E' \rangle = \{\langle e, e' \rangle \mid e \in E \text{ and } e' \in E'\}$ to be the set of all exchanges with an input event from E and output event from E'.

An *execution* or *trace* is a possibly infinite sequence of exchanges. The *empty execution* is an execution that contains no exchanges and is denoted by ϵ. The *length* of an execution x is the number of exchanges in x. The binary *concatenation* operator $\cdot$ is defined for finite-length executions as usual. If x is infinite then $x \cdot y = x$.

Let X and Y be two sets of executions over $\mathcal{E}$. Then XY is the set of all executions obtained by concatenating every execution in X to every execution in Y. More formally, we have $XY = \{xy \mid x \in X, y \in Y\}$. Given a set of executions X, also define $X^0 = \{\epsilon\}$, $X^1 = X$, and $X^n = XX^{n-1}$. Further,

$$X^* = \cup_{i \geq 0} X^i,\ X^+ = \cup_{i>0} X^i,$$
$$X^\omega = \{x_1 x_2 \ldots \mid x_i \in X,\ x_i \neq \epsilon\}\text{, and}$$
$$X^\infty = X^* \cup X^\omega.$$

As usual, we identify a singleton set with the element it contains, e.g. we write $(\langle e, e' \rangle)^\omega$ instead of $(\{\langle e, e' \rangle\})^\omega$. Given a set of executions $X \subseteq \mathcal{E}^\infty$, define X_f to be the set of finite executions in X, i.e. $X_f = X \cap \mathcal{E}^*$.

An execution y is a prefix of an (infinite) execution x if $x = y$ or $x = yx'$ for some (infinite) execution x'. The set of all prefixes of x is denoted by $\mathcal{P}(x)$. We write $y \preceq x$ if y is finite prefix of x, and $y \prec x$ if, in addition, $x \neq y$. The relations $\succ$ and $\succeq$ are defined symmetrically.

Given a set $X \subseteq \mathcal{E}^\infty$, $\mathcal{P}(X)$ is the set of prefixes of every execution in X, i.e. $\mathcal{P}(X) = \cup_{x \in X} \mathcal{P}(x)$. For all executions x and sets of executions X, if $x \in \mathcal{P}(X)$ then we say that x is *alive* in X, or just that x is alive (when X is clear from context). Otherwise we say that x is *dead*.

3 Mandatory Results Automata

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

Definition 1 A *mandatory results automaton* (*MRA*) M is a tuple (E, Q, q_0, δ), where E is the event set over which M operates, Q is a recursively enumerable set of automaton states, q_0 is M's initial state, and $\delta : Q \times E \to Q \times E$ is M's Turing-computable (partial) transition function, which takes M's current state and input event and returns M's next state and output event.

We call q_α a *configuration* of MRA M, where q is M's current state and α is either t or s depending on whether M's next input can come from the target application (t) or the executing system (s). The starting configuration of an MRA is $(q_0)_t$ because the monitor begins executing in its initial state and receives its first input event from the target application.

We define the operational semantics of MRAs with a labeled single-step judgment whose form is $C \xrightarrow{\xi}_M C'$. This judgment indicates that MRA M takes a single step from configuration C to configuration C' while extending the current trace by an exchange ξ. 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. Four inference rules define all possible MRA transitions:

1. *In-Act-Out-Act* enables the MRA to receive a new input action from the target ($next_t$ is the next action generated by the target) and, in response, output an action to the executing system.
2. *In-Act-Out-Res* enables the MRA, immediately after inputting an action a, to return a result r for a to the target.
3. *In-Res-Out-Act* enables the MRA to receive a new input result from the executing system ($next_s$ is

the next result generated by the system) and, in response, output another action to the executing system.

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

Several observations about the operational semantics:

– MRAs can "accept" an input action a by outputting it (with *In-Act-Out-Act*), receiving a result r for a, and then returning r to the application (with *In-Res-Out-Res*).
– MRAs can "halt" an application by outputting an action like `exit`, if the underlying system can execute such an action, or by not outputting events (i.e., leaving the transition function undefined at the point of halting). As in practice, the algorithms implementing MRA transition functions may not always terminate; transition functions are assumed to be Turing computable, not necessarily decidable.
– 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 *In-Act-Out-Res*). 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 *In-Act-Out-Act* and *In-Res-Out-Act*). 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).
– MRAs may 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. Similarly, MRAs have no knowledge of whether and how the target generates actions.

These observations, and the semantics of MRAs in general, match our understanding of how many real program monitors behave. For example, in the Polymer enforcement system [4], 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 [10,11,15].

Limitations Nonetheless, because MRAs are models, some gaps exist between the possible behaviors of MRAs and what real monitors can do in practice. The MRA model has at least four limitations in its applicability to practice, the first three of which are shared by other general enforcement models:

1. MRAs can interpose on and make decisions about *all* security-relevant actions and results, even when such an ability is impractical. For example, monitoring every "clock-tick" action is possible in our model but completely impractical due to performance costs. In practice a monitor would instead request the time from the executing system as needed, or would register a timer with the executing system. Although both of these practical behaviors are also possible in the MRA model, the fact remains that MRAs permit some impractical behaviors.
2. The abstract effects a policy engineer cares about in practice may not align well with a system's concrete interface (i.e., set of events E). For example, a policy engineer may wish to turn garbage collection off at critical points in an execution, but if such an action doesn't exist on the system then the desired effect may not be possible. For MRAs to be effective in practice, the system interface (E) must allow abstract, security-relevant effects to be accurately inferred from, and inserted into, concrete traces.
3. 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).
4. MRAs synchronize (i.e., serialize) security-relevant events as described in Section 1.2, so they cannot be used to secure systems whose correctness would be violated by such synchronization.

Please note the similarity between Limitations 3 and 4: both involve delaying some events and prevent MRAs from being used in cases where such delays cannot be tolerated.

3.1 Example MRAs

We next consider a couple of 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, general models of runtime enforcement, as far as we are aware, because they involve monitors acting on unpredictable results of actions.

Example 1 (Spam-filtering (Result-sanitizing) MRA) In this example, we construct an MRA to secure the interaction of an email client (the target application) with an email server (the executing system). MRA M sanitizes the results of `getMail` actions to filter out spam emails. M's states consist 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 defined by:

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

That is, M outputs its inputs verbatim and does not change its state as long as it does not input a `getMail` action. When M does input `getMail`, it sets its boolean flag and allows `getMail` to execute. If M then inputs a result r for `getMail` (i.e., a list of messages), 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) This 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_d, which must be returned when the user decides not to allow d to execute (r_d 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.

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

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_d; if the user OKs d, M outputs and allows d to execute.

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 that MRAs serve as a good basis for developing a theory of runtime enforcement.

3.2 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 exchange 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 execution x. Formally, judgment $M \Downarrow x$ is defined as follows: for a possibly infinite-length execution $x \in \mathcal{E}^{\infty}$, $M \Downarrow x$ iff there exists a sequence of M-configurations $C_0, C_1, C_2, \ldots$ such that for any n-length prefix x' of x we have

$$C_0 \xrightarrow{\xi_1} C_1 \cdots \xrightarrow{\xi_n} C_n = C_0 \xrightarrow{x'}{}^{*} C_n.$$

The above-mentioned sequence of M-configurations $C_0, C_1, C_2, \ldots$ is called *the run of* M *producing* x. When M is clear from context, the run of M producing x is denoted by $\mathcal{R}_x$. Furthermore, if $C_n = (q)_\alpha$ for some $\alpha \in \{t, s\}$ then we write $q_0 x' = q$. Finally, given an MRA M, *the language* of M, written $\mathcal{L}(M)$, is defined to be $\{x \mid M \Downarrow x\}$.

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 [22]. Policy P is a *property* iff there exists a predicate $\hat{P}$ over $\mathcal{E}^\infty$ such that $\forall X \subseteq \mathcal{E}^\infty : (P(X) \Leftrightarrow \forall x \in X : \hat{P}(x))$. Because there is a one-to-one correspondence between a property P, its predicate $\hat{P}$, and the set of executions that satisfy P (i.e., the set X such that $\forall x \in X : \hat{P}(x)$), the rest of this paper uses P unambiguously to refer to any of the three depending on the context.

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 [9]. For simplicity, 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 3.1 enforces a policy requiring all results of `getMail` actions to be filtered (this policy is a property because it is satisfied iff *every* execution in a set X has exactly zero spam-containing results of `getMail` 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 P could be dissatisfied by execution $\langle \texttt{shutdown}, e\rangle$ (i.e., $\langle \texttt{shutdown}, e\rangle \notin P$) unless $e = \texttt{popupConfirm}$. To enforce this P, an MRA may have no choice but to output `popupConfirm` upon inputting a `shutdown` action. As far as we're aware, policies in existing general models of enforcement cannot specify such relationships between input and output events because the policies are predicates over output executions only.

The primary relationship allowed between input and output events in previous models is *transparency*, which is hardcoded into the definition of enforcement [13,18] and requires 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 verbatim, as described in Section 5), 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). As another example, consider that monitors frequently violate transparency in practice by performing auditing (i.e., modifying the trace to insert privileged logging actions) even when the events being audited are valid. MRA policies can straightforwardly express such non-transparency auditing requirements; on an audited input event e, the policy requires outputting a logging action a, and then when a result r for a is input, the policy requires outputting e (thus building the subtrace $\langle e, a\rangle\langle r, e\rangle$).

In summary, policies here are more expressive than in enforcement models that limit the relationships between monitors' input and output events (e.g., [18,14,6, 7]). To demonstrate the increased expressiveness of the MRA model, Section 5 shows how two particular relationships, transparency and uncontrollable events [3], can be encoded into this paper's framework.

4.2 Enforcement

Unlike previous models of security automata, the current model defines enforcement in terms of soundness and completeness, which are standard principles in the broader fields of security and verification. MRA M is *sound* with respect to property P when M only produces traces satisfying P; M is *complete* with respect to P when it produces all traces satisfying P; and M is *precise* with respect to P when it is sound and complete with respect to P.

Definition 2 MRA M

- *soundly enforces* P iff $\mathcal{L}(M) \subseteq P$,
- *completely enforces* P iff $P \subseteq \mathcal{L}(M)$, and
- *precisely enforces* P iff $\mathcal{L}(M) = P$.

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

As is standard, sound enforcement permits false positives (i.e., false alarms) but not false negatives. Conversely, complete enforcement permits false negatives but not false positives. Precise enforcement permits neither false positives nor false negatives.

For some examples of MRA enforcement, let's consider the policies enforced by the example MRAs from Section 3.1. Because these policies match behaviors of MRAs that automata in previous models cannot mimic, the policies defined below cannot be enforced by previously studied security automata.

Example 3 (Policies enforced by the MRA from Example 1) Consider the spam-filtering MRA M from Example 1. If

– $X = \{\langle \texttt{getMail}, \texttt{getMail}\rangle\langle r, \texttt{filter}(r)\rangle \mid r \in R\}$ and
– $Y = \{\langle a, a\rangle\langle r, r\rangle \mid a \in A, r \in R, \text{ and } a \neq \texttt{getMail}\}$,

then the set of the executions produced by M is $P = \mathcal{P}((X \cup Y)^\infty)$, so M precisely enforces P, completely enforces every subset of P, and soundly enforces every superset of P.

Example 4 (Policies enforced by the MRA from Example 2) Now let M be the dangerous-action-confirming MRA from Example 2. If

– $X = \{\langle d, \texttt{popupConfirm}\rangle\langle \texttt{OK}, d\rangle\langle r, r\rangle \mid r \in R\}$,
– $Y = \{\langle d, \texttt{popupConfirm}\rangle\langle \texttt{cancel}, r_d\rangle\}$, and
– $Z = \{\langle a, a\rangle\langle r, r\rangle \mid a \in A,\ r \in R, \text{ and } a \neq d\}$,

then the set of executions produced by M is $P = \mathcal{P}((X \cup Y \cup Z)^\infty)$. Again, M precisely enforces P, completely enforces every subset of P, and soundly enforces every superset of P.

5 Analysis of MRA-enforceable Policies

This section analyzes the properties that MRAs can enforce soundly, completely, and precisely. Throughout this section, and for the remainder of the paper, we assume that we're dealing with a system having action set A, result set R, and event set E.

The following definition and lemma describe the structure of executions that can be observed during a run of an MRA. Executions are constrained in part because MRAs may only input a result (action) after outputting an action (result or nothing, initially).

Definition 3 The set

$$\mathcal{P}((\langle A, R\rangle^\infty \langle A, A\rangle \langle R, A\rangle^\infty \langle R, R\rangle)^\infty)$$

is called the *MRA execution universe* and is denoted by U.

MRAs can only produce executions in U.

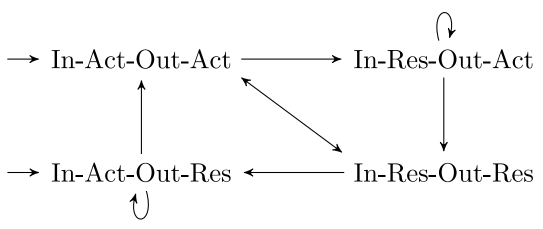

Fig. 3 The order in which Figure 2's rules may be applied.

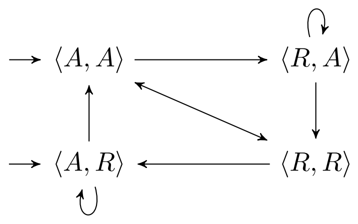

Fig. 4 A graph whose paths correspond to executions in U.

Lemma 1 $\forall M : \mathcal{L}(M) \subseteq U$

Proof The claim is a direct consequence of MRAs' single-step semantics. The initial transition of an MRA must be either In-Act-Out-Act or In-Act-Out-Res, and subsequent transitions must be ordered as described in Figure 3. The executions produced by making transitions in the orders depicted in Figure 3 correspond to paths in the graph depicted in Figure 4, which in turn correspond to executions satisfying the ∞-regular expression defining U. Hence, every trace built by an MRA must be an element of U. □

Definition 4 presents some interesting classes of properties, including exchange-based versions of safety [17] and liveness [2].

Definition 4 A property P is called

– *exchange prefix closed*, when
$\forall x \in \mathcal{E}^\infty : (x \in P \Rightarrow \mathcal{P}(x) \subseteq P)$
(or equivalently $\forall x \in \mathcal{E}^\infty : (x \in P \Rightarrow \mathcal{P}(x)_f \subseteq P)$).
– *exchange omega closed*, when
$\forall x \in \mathcal{E}^\omega : (\mathcal{P}(x)_f \subseteq P \Rightarrow x \in P)$
(or equivalently $\forall x \in \mathcal{E}^\infty : (\mathcal{P}(x)_f \subseteq P \Rightarrow x \in P)$).
– *exchange safety*, when
$\forall x \in \mathcal{E}^\infty : (x \notin P \Rightarrow \exists x' \preceq x : \forall y \succeq x' : y \notin P)$.
– *MRA liveness*, when
$\forall x \in U_f : \exists y \succeq x : y \in P$.

Let PC be the set of all exchange-prefix-closed properties, OC the set of exchange-omega-closed properties, ES the set of exchange-safety properties, and ML the set of MRA-liveness properties.

The following lemma shows that safety properties can alternatively be defined as the intersection of the prefix- and omega-closed properties. That is, P is a safety property iff $\forall x \in \mathcal{E}^\infty : (x \in P \Leftrightarrow \mathcal{P}(x)_f \subseteq P)$.

Lemma 2 *ES* $=$ *PC* $\cap$ *OC*

Proof ($\subseteq$) Let $P \in$ ES be an exchange-safety property. We show that $P \in$ PC. Assume that $x \in P$, but $\mathcal{P}(x) \nsubseteq P$. Then there exists a prefix x' of x such that $x' \notin P$. Because $P \in$ ES, there is $x'' \preceq x'$ such that $\forall y \succeq x''$, $y \notin P$. Notice that $x \succeq x''$, and so $x \notin P$ which is a contradiction. This shows that $P \in$ PC.

We next show that $P \in$ OC. Assume that for some $x \in \mathcal{E}^\omega, \mathcal{P}(x)_f \subseteq P$ but $x \notin P$. Then because P is in ES, there exists $x' \preceq x$ such that $x' \notin P$, which contradicts the assumption that $\mathcal{P}(x)_f \subseteq P$. This shows that $P \in$ OC.

($\supseteq$) Let $P \in$ PC $\cap$ OC. Assume that $x \in \mathcal{E}^\infty$ and $x \notin P$. If $x \in \mathcal{E}^*$, then for all $y \succeq x$, $y \notin P$; if a $y \succeq x$ were in P, then because $P \in$ PC, we would have $x \in P$ (a contradiction). On the other hand, if $x \in \mathcal{E}^\omega$, then because $P \in$ OC and $x \notin P$, $\exists x' \preceq x$ such that $x' \notin P$, so by the same reasoning used in the previous sentence, $\forall y \succeq x'$, $y \notin P$. Hence, $P \in$ ES. □

As in previous enforcement models, the execution universe here is a safety property.

Lemma 3 $U \in$*ES*

Proof The traces of MRA execution universe U correspond to possibly infinite paths in the graph G depicted in Figure 4. U is prefix closed because for every path x in G, all prefixes of x must also be paths in G. U is omega closed because for all infinite-length executions x, if all the finite prefixes of x are paths in G then so must be x. U is therefore in PC $\cap$ OC, which equals ES by Lemma 2. □

Let MS (MC, MP) denote the set of security properties soundly (completely, precisely) enforceable by MRAs. To establish succinct characterizations of MS, MC, and MP, we define two additional sets of properties as follows.

Definition 5 A property P is called *reasonable* when (a) $\epsilon \in P$, (b) $P \subseteq U$, and (c) $\mathcal{P}(P)_f$ is a recursively enumerable set. The set of all reasonable properties is denoted by RS.

Definition 6 A property P is called *deterministic* when, for all finite-length executions x and events e, there exists at most one exchange $\xi = \langle e, e' \rangle$ such that $x\xi$ is alive in P. The set of all deterministic properties is denoted by DT.

The next theorem shows that MRAs precisely enforce exactly those properties that are reasonable, deterministic, and exchange safety. The proof is constructive; it shows how to create an MRA to precisely enforce any such property. For example, given the spam-filtering property defined in Example 3 of Section 4.2, the proof constructs an MRA $M = (E, \mathcal{E}^*, \epsilon, \delta)$, where $\delta(q, e)$ is defined to be $(q\langle e,\texttt{filter}(e)\rangle, \texttt{filter}(e))$ when q ends with $\langle\texttt{getMail},\texttt{getMail}\rangle$ and $(q\langle e, e\rangle, e)$ otherwise. This M precisely enforces P and is functionally equivalent to the spam-filtering MRA defined in Example 1 of Section 3.1.

Theorem 1 *MP* $=$ *RS* $\cap$ *DT* $\cap$ *ES*

Proof ($\subseteq$) Suppose that P is precisely enforceable and let $M = (E, Q, q_0, \delta)$ be an MRA with $\mathcal{L}(M) = P$.

– *P is reasonable.* If the target never outputs an action, then M never makes a transition and so produces the empty execution. This shows that ϵ is always in $\mathcal{L}(M) = P$. Also, according to Lemma 1, $P \subseteq U$. Finally, $\mathcal{P}(P)_f = \mathcal{P}(\mathcal{L}(M))_f$ can be enumerated because (1) the events in E can be enumerated, and (2) δ is Turing-computable.
– *P is deterministic.* For all finite-length executions x and events e and e', if $q = q_0 x$ and $x\langle e, e'\rangle$ is alive in P, then $\delta(q, e) = (q', e')$ (for some state q'), implying that P is deterministic.
– *P is exchange-safety.* $\mathcal{L}(M) = P$ is in PC because for all executions x in $\mathcal{L}(M)$, M produces all prefixes of x on its way to producing x itself. $\mathcal{L}(M) = P$ is also in OC because M is deterministic, so if M produces all the finite prefixes of an infinite execution x, then M must also produce x (because M, being deterministic, always passes through the same configurations used to produce x' when producing x, if $x' \preceq x$). Hence, P is in PC $\cap$ OC $=$ ES.

($\supseteq$) Consider a property P that is reasonable, deterministic, and exchange-safety (meaning prefix and omega closed). Let $M = (E, \mathcal{E}^*, \epsilon, \delta)$ be an MRA with $\delta(q, e) = (q\langle e, e'\rangle, e')$ if $q\langle e, e'\rangle$ is alive in P. Observe that δ is a Turning-computable function because P is reasonable, prefix closed, and deterministic. Thus M is well defined. Next we show that $\mathcal{L}(M) = P$.

First observe that $\mathcal{L}(M) \subseteq P$ because (1) the definition of δ ensures that all finite executions produced by M are alive in P, (2) P is prefixed closed, and (3) P is omega closed.

To see that $P \subseteq \mathcal{L}(M)$, first consider an infinite execution $x = \xi_0\xi_1\xi_2\ldots$ in P. For all $n \geq 0$, let $C_n = (x_n)_\alpha$, where (1) x_n is the n-length prefix of x, and (2) α is t if ξ_n begins with an action and s otherwise. Because $\forall n \geq 0 : C_n \xrightarrow{\xi_n} C_{n+1}$ (by the definition of δ

and the fact that $P \subseteq U$), we have that M produces x, so $x \in \mathcal{L}(M)$. A similar argument shows that all finite x in P are also in $\mathcal{L}(M)$, so $P \subseteq \mathcal{L}(M)$.

□

The next corollary shows that a property P is soundly enforceable by an MRA iff P contains ϵ.

Corollary 1 $MS = \{P \mid \epsilon \in P\}$

Proof ($\subseteq$) If P is soundly enforceable, then there exists an MRA M such that $\mathcal{L}(M) \subseteq P$. By Theorem 1, $\mathcal{L}(M)$ is reasonable, so $\epsilon \in \mathcal{L}(M)$. Hence, $\epsilon \in P$ as required.

($\supseteq$) Consider a property P containing ϵ. Observe that the property $\{\epsilon\}$ is reasonable, deterministic, and exchange safety, and hence precisely enforceable by Theorem 1. Because every superset of a precisely enforceable property is soundly enforceable, P is soundly enforceable. □

In terms of complete enforcement, a property P is enforceable by MRAs iff P is a subset of a reasonable and deterministic property.

Corollary 2 $MC = \{P \mid \exists P' \supseteq P : P' \in RS \cap DT\}$

Proof ($\subseteq$) If P is completely enforceable, then there exists an MRA M such that $P \subseteq \mathcal{L}(M)$. Note that by Theorem 1, $\mathcal{L}(M)$ is deterministic and reasonable.

($\supseteq$) Let P be a subset of a deterministic and reasonable property P'. Define

$$\hat{P} = \mathcal{P}(P') \cup \{x \in \mathcal{E}^\omega \mid \mathcal{P}(x)_f \subseteq \mathcal{P}(P')\}.$$

Note that if $\hat{P}$ is precisely enforceable, then P is completely enforceable because $P \subseteq P' \subseteq \hat{P}$. Thus we finish the proof by showing that $\hat{P}$ is deterministic, reasonable, and exchange safety, and hence precisely enforceable by Theorem 1.

– $\hat{P}$ *is deterministic.* Suppose that for a finite execution x and an event e, there are two events e', e'' such that $x\langle e, e'\rangle$ and $x\langle e, e''\rangle$ are alive in $\hat{P}$. By the definition of $\hat{P}$, these executions are prefixes of executions in P' and so are alive in P'. Because P' is deterministic, $e' = e''$. This proves determinism of $\hat{P}$.
– $\hat{P}$ *is reasonable.* First note that $\hat{P}$ is a superset of P', which is reasonable and therefore contains ϵ. In addition, $P' \subseteq U$, and because U is omega closed (by Lemmas 2–3), $\{x \in \mathcal{E}^\omega \mid \mathcal{P}(x)_f \subseteq U\} \subseteq U$. Using these observations and the definition of $\hat{P}$,

$$\begin{aligned}\hat{P} &= \mathcal{P}(P') \cup \{x \in \mathcal{E}^\omega \mid \mathcal{P}(x)_f \subseteq \mathcal{P}(P')\}\\ &\subseteq \mathcal{P}(U) \cup \{x \in \mathcal{E}^\omega \mid \mathcal{P}(x)_f \subseteq \mathcal{P}(U)\}\\ &\subseteq U \cup \{x \in \mathcal{E}^\omega \mid \mathcal{P}(x)_f \subseteq U\}\\ &\subseteq U.\end{aligned}$$

Finally, the set $\mathcal{P}(\hat{P})_f$ equals $\mathcal{P}(P')_f$ and so is recursively enumerable.
– $\hat{P}$ *is exchange safety.* $\hat{P}$ is by definition prefix closed and omega closed, so by Lemma 2, $\hat{P} \in$ ES.

□

Next let's consider two specific requirements one may want to place on how MRAs transform input into output events. The first is transparency [18], which requires that valid traces input to a monitor not be modified. Although we've come to believe that transparency is often overly restrictive (as discussed in Section 4.1), we can incorporate transparency into the MRA model.

In the context of transparency, some input traces of the form $a_1; r_1; a_2; r_2; \ldots$ are considered valid. Let I be the set of such valid input traces. For an MRA to be transparent, it must output all elements of I verbatim. That is, for all $a_1; r_1; a_2; r_2; \ldots \in I$, the MRA must produce the trace $\langle a_1, a_1\rangle\langle r_1, r_1\rangle\langle a_2, a_2\rangle\langle r_2, r_2\rangle \ldots$. Let V_I be the set of all traces formed by verbatim expanding I in this way. Formally, V_I is

$$\{\langle a_1, a_1\rangle\langle r_1, r_1\rangle\langle a_2, a_2\rangle\langle r_2, r_2\rangle \ldots \mid a_1; r_1; a_2; r_2; \ldots \in I\}.$$

Then for a given I, the set of properties precisely and transparently enforceable by MRAs is

$$\mathrm{MP} \cap \{P \mid V_I \subseteq P\}$$

because the properties must require that every valid input trace be output verbatim. Trivially, when $I = \emptyset = V_I$ the set of properties precisely and transparently enforceable by MRAs remains MP.

Uncontrollable events [3] induce a similar requirement as transparency, in that uncontrollable events must be output verbatim. A major difference between transparency and uncontrollable events is that transparency restricts outputs execution-wide, while uncontrollable events restrict the output of events in isolation. Let $W \subseteq E$ be the set of uncontrollable events; on any input $e \in W$ during an MRA's operation, the MRA must output e. Also let $\mathcal{E}_W$ be the set of exchanges $\{\langle e, e\rangle \mid e \in W\}$, and W^C be the set of controllable events (i.e., $W^C = E \setminus W$). Then the set of properties precisely enforceable by MRAs is

$$\mathrm{MP} \cap \{P \mid P \subseteq (\mathcal{E}_W \mid \langle W^C, E\rangle)^\infty\}.$$

To prevent monitors from "manufacturing" uncontrollable events, one might wish to further restrict MRAs to output only uncontrollable events that have just been input. In this case, the set of properties precisely enforceable by MRAs is

$$\mathrm{MP} \cap \{P \mid P \subseteq (\mathcal{E}_W \mid \langle W^C, W^C\rangle)^\infty\}.$$

Similar to the trivial case of transparency, when $W = \emptyset$ the set of properties precisely enforceable remains MP.

6 Nondeterministic MRAs

Because the behavior of an MRA is determined by its transition function, MRAs model deterministic security monitors. Although this is an adequate constraint for most real-life security monitors, there are situations when it is beneficial to consider some redundancy (i.e., nondeterminism) in properties and monitors. Such situations arise naturally when external factors may influence whether an execution is valid, or how a monitor behaves. For example, a multithreaded monitor's behavior may be subject to an external thread scheduler; the properties enforceable by multithreaded monitors may allow for such nondeterminism. As another example, monitors may be influenced by auxiliary, unpredictable inputs, such as readings from weather sensors or spontaneous human input.

The aim of this section is to introduce a model of generic nondeterministic security monitors and investigate the properties they can enforce.

Definition 7 A *nondeterministic MRA* (NMRA) N is a tuple (E, Q, I, δ), where E is the event set over which N operates, Q is a recursively enumerable set of automaton states, $I \subseteq Q$ is a recursively enumerable set of N's initial states, and $\delta \subseteq Q \times E \times Q \times E$ is a recursively enumerable transition relation.

The single-step ($\rightarrow$), multi-step ($\rightarrow^*$) and production ($\Downarrow$) relations are defined for NMRAs in the same ways as for MRAs (though for NMRAs, the single-step rules of Figure 2 would have to replace premises of the form $\delta(q, e) = (q', e')$ with premises of the form $(q, e, q', e') \in \delta$).

Let NS (NC, NP) denote the collection of security properties soundly (completely, precisely) enforced by NMRAs. We now investigate the security properties NMRAs soundly, completely, and precisely enforce.

First, as with MRAs, NMRAs can only produce traces in the universe U.

Lemma 4 $\forall N : \mathcal{L}(N) \subseteq U$

Proof Observe that the proof of Lemma 1 is applicable to NMRAs, and so also proves the claim of this lemma. □

We next consider NMRAs as precise enforcers and find that for every reasonable, prefix- and omega-closed property P, there exists an NMRA that precisely enforces P. Moreover, if there exists an NMRA that precisely enforces P, then P is reasonable and prefix closed. These inclusions are strict, so we can write this result formally as (RS ∩ PC ∩ OC) ⊂ NP ⊂ (RS ∩ PC). By Lemma 2, an equivalent way to state this result is as follows.

Theorem 2 *RS ∩ ES ⊂ NP ⊂ RS ∩ PC*

Proof We start by proving RS ∩ ES ⊆ NP. Let P be a property in RS ∩ ES, T be a Turing Machine that enumerates $\mathcal{P}(P)_f$ (which must exist because $P \in$ RS), and $N = (E, Q, I, \delta)$ be an NMRA such that

- $Q = \{x_i \mid x \in \mathcal{E}^* \text{ and } i \in \mathbb{N}\}$,
- $I = \{\epsilon_i \mid i \in \mathbb{N}\}$, and
- δ is defined as follows. Given x_i and e, run T for i steps. If the final execution output by T is $x\langle e, e'\rangle$, for some event e', then $(x_i, e, x\langle e, e'\rangle_j, e') \in \delta$, for all $j \in \mathbb{N}$.

Note that δ's definition implies that it is recursively enumerable, so N is well defined. Next we show that $\mathcal{L}(N) = P$.

First observe that $\mathcal{L}(N) \subseteq P$ because (1) the definition of δ ensures that all finite executions produced by N are alive in P, (2) P is prefixed closed, and (3) P is omega closed.

To see that $P \subseteq \mathcal{L}(N)$, first consider an infinite execution $x = \xi_0\xi_1\xi_2\ldots$ in P. For all $n \geq 0$, let $C_n = ((x_n)_i)_\alpha$, where (1) x_n is the n-length prefix of x, (2) i is such that T outputs the $(n+1)$-length prefix of x after i iterations, and (3) α is t if ξ_n begins with an action and s otherwise. Because $\forall n \geq 0 : C_n \xrightarrow{\xi_n} C_{n+1}$ (by the definition of δ and the fact that $P \subseteq U$), we have that N produces x, so $x \in \mathcal{L}(N)$. A similar argument shows that all finite x in P are also in $\mathcal{L}(N)$, so $P \subseteq \mathcal{L}(N)$. This completes the proof that RS ∩ ES ⊆ NP.

To prove that NP ⊆ RS ∩ PC, suppose that P is precisely enforceable and let $N = (E, Q, I, \delta)$ be an NMRA with $\mathcal{L}(N) = P$.

- *P is reasonable.* If the target never outputs an action, then N never makes a transition and so produces the empty execution. This shows that ϵ is always in $\mathcal{L}(N) = P$. Also, according to Lemma 4, $P \subseteq U$. Finally, $\mathcal{P}(P)_f = \mathcal{P}(\mathcal{L}(N))_f$ can be enumerated because (1) the events in E can be enumerated, and (2) δ is recursively enumerable.
- *P is prefix closed.* $\mathcal{L}(N) = P$ is in PC because for all executions x in $\mathcal{L}(N)$, N produces all prefixes of x on its way to producing x itself.

We now prove that both inclusions are strict. First we give an example of a property $\bar{P}$ precisely enforceable by an NMRA that is not exchange-safety. Define $N = (E, \mathbb{N}, \mathbb{N}, \delta)$ such that $\delta = \{(i, e, i{-}1, e') \mid e, e' \in E \text{ and } i > 1\}$. Then $\bar{P} = \mathcal{L}(N) = U_f$, but U_f is not omega closed and therefore not in ES.

Finally, because (1) RS ∩ PC is uncountable, and (2) NP is countable, we have that (RS ∩ PC) \ NP is nonempty (and uncountable). (1) holds because properties in RS ∩ PC need not be omega closed and may therefore contain arbitrary subsets of infinite-length executions (as long as the subsets don't violate reasonableness or prefix closure). (2) holds because the set of NMRAs is countable (events, states, initial states, and transitions are all recursively enumerable). □

As a corollary of Theorem 2, there exists an NMRA that produces U. In contrast, although all MRAs produce subsets of U, no MRA can produce U itself (because U is nondeterministic).

Corollary 3 $\exists N : \mathcal{L}(N) = U$

Proof U is reasonable (by the definitions of reasonable and U) and exchange safety (by Lemma 3), so U is precisely enforceable by an NMRA (by Theorem 2).

As with MRAs, a property P is soundly enforceable by an NMRA iff P contains ϵ.

Corollary 4 $NS = \{P \mid \epsilon \in P\}$

Proof ($\subseteq$) If P is soundly enforceable, then there exists an NMRA N such that $\mathcal{L}(N) \subseteq P$. By Theorem 2, $\mathcal{L}(N)$ is reasonable, so $\epsilon \in \mathcal{L}(N)$. Hence, $\epsilon \in P$ as required.

($\supseteq$) If $\epsilon \in P$, then by Corollary 1, $P \in$ MS. Because every MRA is an NMRA, P is also in NS. □

In terms of complete enforcement, a property P is enforceable by NMRAs iff P is a subset of a reasonable property.

Corollary 5 $NC = \{P \mid \exists P' \supseteq P : P' \in RS\}$

Proof ($\subseteq$) If P is completely enforceable, then there exists an NMRA N such that $P \subseteq \mathcal{L}(N)$. Note that by Theorem 2, $\mathcal{L}(N)$ is reasonable.

($\supseteq$) Assume that P is a subset of a reasonable property and is therefore also a subset of U. By Corollary 3, there exists an NMRA that precisely enforces U, so that same NMRA completely enforces P. □

7 A Hierarchy of Reasonable Properties

This section ties together the results of previous sections, to establish relationships between various classes of reasonable properties. In particular, Corollaries 6–8 cast the results of earlier theorems and corollaries into the domain of the reasonable properties.

For notational convenience, let RXX denote RS $\cap$ XX, where XX is a class of properties. For example, RMS is RS $\cap$ MS, the set of reasonable properties that can be soundly enforced by MRAs.

Corollary 6 *RS=RMS=RNS=RNC and RMC=RDT*

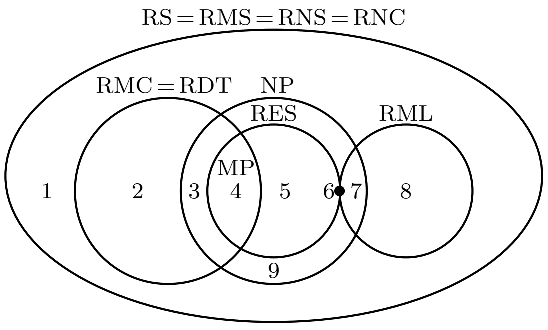

Fig. 5 A hierarchy of reasonable security properties.

Proof Because all reasonable properties contain ϵ, Corollaries 1 and 4 imply that RS = RMS = RNS. Also, Corollary 5 implies that RS $\subseteq$ NC, so RS = RNC as well. Finally, Corollary 2 implies that RDT $\subseteq$ MC $\subseteq$ DT (the latter inclusion because subsets of deterministic properties must be deterministic), so RMC = RDT. □

Corollary 7 *MP=RDT$\cap$ RES and RES$\subset$NP$\subset$RPC*

Proof Immediate by Theorems 1–2. □

Corollary 8 *RMC$\cap$ RML=$\emptyset$ and RES$\cap$ RML=$\{U\}$*

Proof As shown in the proof of Corollary 6, MC $\subseteq$ DT. Also note that for all properties $P \in$ ML, *all* executions in U_f must be alive in P, implying that DT $\cap$ ML = $\emptyset$. Hence, MC$\cap$ML = $\emptyset$, so RMC$\cap$RML = $\emptyset$. In addition, if $P \in$ ES $\cap$ ML, then every execution in U_f must be alive in P, yet P must be prefix closed (hence $U_f \subseteq P$) and omega closed (hence $U \subseteq P$). The only reasonable property P such that $U \subseteq P$ is U itself, so RES$\cap$RML = $\{U\}$. □

Figure 5 summarizes the results of Corollaries 6–8. The nine example properties depicted in Figure 5 can be defined as follows.

Property 1. Property 1 could be any nondeterministic, non-prefix-closed, non-liveness property. For example, consider the following sets of executions.

- X=$\{\langle d, \texttt{popupConfirmError}\rangle\langle \texttt{OK}, d\rangle\langle r, r\rangle \mid r{\in}R\}$
- Y=$\{\langle d, \texttt{popupConfirmError}\rangle\langle \texttt{cancel}, r_d\rangle\}$
- X'=$\{\langle d, \texttt{popupConfirmWarning}\rangle\langle \texttt{OK}, d\rangle\langle r, r\rangle \mid r{\in}R\}$
- Y'=$\{\langle d, \texttt{popupConfirmWarning}\rangle\langle \texttt{cancel}, r_d\rangle\}$

Let $P = (X \cup Y \cup X' \cup Y')^\infty$. This property allows monitors to respond to dangerous events by confirming them with either error or warning messages; according to the property, users must respond to the confirmation messages. Note that P is reasonable but nondeterministic (because both $\langle d, \texttt{popupConfirmError}\rangle$ and

$\langle d, \texttt{popupConfirmWarning}\rangle$ are alive), non-prefix-closed (because $\langle d, \texttt{popupConfirmError}\rangle$ is invalid but alive), and non-liveness (because execution $\langle d, d\rangle$ is not alive).

Property 2. Reusing the definitions of X and Y from Property 1, let $P = (X \cup Y)^\infty$. As before, P is reasonable, non-prefix-closed, and non-liveness. However, P is now deterministic because the choice of error versus warning messages has been removed.

Property 3. Consider the NMRA N used to define $\bar{P}$ in Theorem 2 (i.e., $\mathcal{L}(N) = U_f$). Removing from N's transition relation all tuples of the form $(i, e, i{-}1, e')$ such that $e \neq e'$ guarantees that $\mathcal{L}(N)$ is (1) deterministic (because there is exactly one valid output for every input), (2) in NP (because N is an NMRA), (3) not in RES (because $\mathcal{L}(N)$ remains non-omega-closed), and (4) not in RML (because the execution $\langle e, e'\rangle$, where $e \neq e'$, is dead).

Property 4. We have already seen properties in MP: Examples 3 and 4 defined properties precisely enforced by the MRAs constructed in Examples 1 and 2.

Property 5. Property 5 could be any nondeterministic exchange-safety property. For example, $P = \{\epsilon, \langle \texttt{fopen}, \texttt{fopen}\rangle, \langle \texttt{fopen}, \texttt{logfopen}\rangle\}$ is in RES $\setminus$ MP.

Property 6. Per Corollary 8, Property 6 is U.

Property 7. Property 7 could be U_f, which is a nondeterministic, non-safety, liveness property that is in NP (as shown in the proof of Theorem 2).

Property 8. Property 8 could be any non-prefix-closed liveness property, e.g., a minimum-activity property forbidding exactly those executions of length between 1 and i (for some fixed $i{>}0$).

Property 9. Returning to the NMRA N used to define $\bar{P}$ in Theorem 2 (i.e., $\mathcal{L}(N) = U_f$), remove from N's transition relation all tuples of the form $(i, e, i{-}1, e)$. Then assuming that E contains multiple actions and results, $\mathcal{L}(N)$ is nondeterministic, non-safety (because $\mathcal{L}(N)$ remains non-omega-closed), and non-liveness (because $\langle e, e\rangle$ is dead), but still in NP (because N is an NMRA).

8 Discussion

To summarize the enforcement capabilities of MRAs and NMRAs:

– MRAs precisely enforce deterministic properties that are reasonable and prefix and omega closed, while NMRAs can precisely enforce all properties that are reasonable and prefix and omega closed. NMRAs can also precisely enforce countably many of the uncountable properties that are reasonable and prefix closed but not omega closed.
– Both MRAs and NMRAs soundly enforce all reasonable properties.
– Although NMRAs completely enforce all reasonable properties, MRAs completely enforce only the deterministic reasonable properties.

In terms of (reasonable) safety and liveness properties, we've found that MRAs precisely enforce a strict subset of safety properties (i.e., the ones that are also deterministic), while NMRAs precisely enforce a strict superset of safety properties. However, this analysis would change under alternative definitions of safety and liveness. For example, one could define safety and liveness on an event-by-event basis, rather than the exchange-based definitions used in this paper. Under an event-based definition of safety, the property $P = \{\epsilon, \langle a, a\rangle\}$ could be considered non-safety because it specifies that just inputting a, without also outputting a, is invalid but can be rectified by outputting a. MRAs can precisely enforce event-based non-safety properties like this P, as long as the non-safety is always within exchanges.

MRAs can also enforce nonsafety properties in cases where we assume that the executing system *will* return results (at least for some actions). Currently, MRAs enforce only exchange-safety properties because they output at most one event in response to every input, which may or may not appear. Hence, sound MRAs must ensure the validity of their trace after every output. Edit automata, on the other hand, can enforce nonsafety properties by outputting a sequence of actions σ in response to a single input; the trace may be invalid partway through σ but become valid by the end. MRAs could achieve the same effect, and thereby enforce nonsafety properties as well, under the assumption that the executing system returns results for actions it receives. Then MRAs could, for example, output a temporarily invalid action and receive its result, and then output a valid action and receive its result, thus enforcing a non-exchange-safety property. The assumption that executing systems will return results for actions is often reasonable; CPUs, operating systems, and virtual machines are often designed to guarantee that a result, possibly indicating a timeout, eventually gets returned for many or all actions.

More important than the low-level analysis of properties enforceable by MRAs are the higher-level ideas of including monitors' input and output events on traces and defining enforcement in terms of soundness and completeness. With these techniques, policies become monitor centric (i.e., they specify valid/invalid monitor behaviors), versus the target-centric policies of earlier security-automata and formal-verification frameworks (which specify valid/invalid target behaviors). In the domain of runtime enforcement, monitor-centric policies have two primary advantages over target-centric policies: (1) monitor-centric policies are more expressive because they can require arbitrary relationships to hold between monitors' inputs and outputs, and (2) monitor-centric policies simplify definitions of enforcement because the definitions no longer have to hardcode particular input-output relationships.

Acknowledgments

We're grateful to the anonymous reviewers for their many helpful comments. This research was supported by US National Science Foundation grants CNS-0716343, CNS-0831785, 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. D. Basin, V. Jugé, F. Klaedtke, and E. Zălinescu. Enforceable security policies revisited. *ACM Transactions on Information and System Security*, 16(1), June 2013.
4. L. Bauer, J. Ligatti, and D. Walker. Composing expressive runtime security policies. *ACM Transactions on Software Engineering and Methodology*, 18(3):1–43, 2009.
5. D. Beauquier, J. Cohen, and R. Lanotte. Security policies enforcement using finite edit automata. *Electron. Notes Theor. Comput. Sci.*, 229(3):19–35, 2009.
6. N. Bielova and F. Massacci. Predictability of enforcement. In *Proceedings of the International Symposium on Engineering Secure Software and Systems 2011*, volume 6542, pages 73–86. Springer, 2011.
7. N. Bielova and F. Massacci. Iterative enforcement by suppression: Towards practical enforcement theories. *Journal of Computer Security*, 20(1):51–79, 2012.
8. 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.
9. 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.
10. Ú. Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Cornell University, Jan. 2004.
11. J. Finnis, N. Saigal, A. Iamnitchi, and J. Ligatti. A location-based policy-specification language for mobile devices. *Pervasive and Mobile Computing Journal*, 8(3):402–414, June 2012.
12. P. W. L. Fong. Access control by tracking shallow execution history. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2004.
13. K. Hamlen, G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *ACM Transactions on Programming Languages and Systems*, 28(1):175–205, Jan. 2006.
14. R. Khoury and N. Tawbi. Corrective enforcement of security policies. *Formal Aspects of Security and Trust*, pages 176–190, 2010.
15. 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.
16. 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.
17. L. Lamport. Logical foundation. *Lecture Notes in Computer Science*, 190:119–130, 1985.
18. 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.
19. J. Ligatti and S. Reddy. A theory of runtime enforcement, with results. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, Sept. 2010.
20. A. Pretschner, M. Buechler, M. Harvan, C. Schaefer, and T. Walter. Usage control enforcement with data flow tracking for x11. In *Proc. 5th Intl. Workshop on Security and Trust Management*, pages 124–137, 2009.
21. A. Pretschner, M. Hilty, D. Basin, C. Schaefer, and T. Walter. Mechanisms for usage control. In *Proceedings of the 2008 ACM symposium on Information, computer and communications security*, 2008.
22. F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and Systems Security*, 3(1):30–50, Feb. 2000.
23. C. Talhi, N. Tawbi, and M. Debbabi. Execution monitoring enforcement under memory-limitation constraints. *Information and Computation*, 206(2–4):158–184, 2008.
24. M. Viswanathan. *Foundations for the Run-time Analysis of Software Systems*. PhD thesis, University of Pennsylvania, 2000.
25. 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 Summary of Symbols

Symbol	Description
Σ	finite alphabet
a	an action
A	set of actions
r	a result
R	set of results
e	an event
E	set of events
ξ	an exchange
$\mathcal{E}$	set of exchanges
ϵ	empty execution
x, y	executions
X, Y	sets of executions
$\mathcal{P}(X)$	set of prefixes of executions in X
X_f	set of finite executions in X
M	mandatory results automaton (MRA)
N	nondeterministic MRA
T	Turing machine
α	either t (the target) or s (the system)
next_α	next input from α
$\mathcal{R}_x$	run (i.e., sequence of configurations) producing x
P	policy/property
U	MRA execution universe
MS	properties soundly enforceable by MRAs
MC	properties completely enforceable by MRAs
MP	properties precisely enforceable by MRAs
NS	properties soundly enforceable by NMRAs
NC	properties completely enforceable by NMRAs
NP	properties precisely enforceable by NMRAs
PC	exchange-prefix-closure properties
OC	exchange-omega-closure properties
ES	exchange-safety properties
ML	MRA-liveness properties
RS	reasonable properties
DT	deterministic properties
RXX	reasonable properties in set XX, that is, $\text{RS} \cap XX$

Table 1 Symbols and their meaning.