

A Theory of Runtime Enforcement, with Results*

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

October 28, 2009

Abstract

In many real systems, security-relevant actions produce security-relevant results, and an application attempting to execute an action may not be able to make progress until it receives a result for that action. However, current models of runtime enforcement are limited to reasoning about policies and mechanisms in terms of actions alone, without considering the results of those actions. Hence, a fundamental gap exists between real enforcement systems and models of those systems. This paper bridges the action-results gap by presenting a more general model of runtime enforcement in which all actions return (possibly void- or unit-type) results and applications require results from actions in order to make progress. This new mandatory-results model more accurately reflects the capabilities and limitations of real runtime monitors. We analyze the new model and show that if (1) applications may terminate without monitors being notified and (2) systems that execute actions may never return results for those actions, then monitors can enforce exactly the set of safety properties satisfied by empty executions. However, systems are often built to guarantee that scenarios (1) and (2) above cannot occur, in which case we show that runtime monitors can enforce a succinctly specifiable set of policies we call the action-life properties. Besides the safety properties, action-life properties include some nonsafety liveness and nonsafety nonliveness properties.

1 Introduction

Runtime enforcement mechanisms dynamically monitor untrusted applications to ensure that they obey desired policies. Although computability and proof theory provide good frameworks for understanding which policies can be enforced through static analysis, we currently lack sufficiently general frameworks for understanding which policies can be enforced through runtime mechanisms. This paper presents a new, general framework for reasoning about runtime enforcement and investigates the limits of enforcing policies dynamically, an important problem given the popularity of runtime security mechanisms (as in firewalls, operating systems, auditing tools, spam filters, intrusion-detection systems, access-control systems, stack inspection, etc).

Much research has been performed to model generic runtime mechanisms and analyze their enforcement capabilities (cf. Section 6). However, existing models of generic runtime mechanisms are based on the system abstractions shown in Figure 1. Figure 1a depicts a target application issuing instructions—or more abstractly, *actions*—for an underlying system (such as an operating system, virtual machine, or CPU) to execute. We can secure the application by interposing a monitoring mechanism between the application and the underlying system, as in Figure 1b. The mechanism, a security monitor, dynamically transforms the application’s actions to ensure that the overall sequence of actions actually executed is *valid* (i.e., *satisfies* a desired policy). The monitor may or may not be inlined into the target application.

*Technical Report USF-CSE-SS-102809

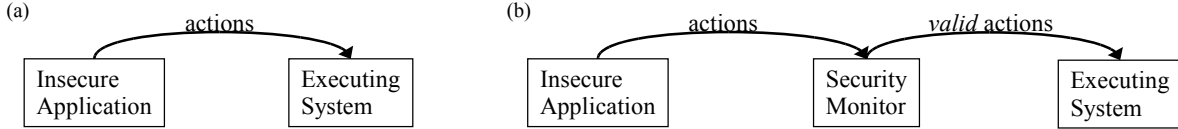


Figure 1: Existing model of an insecure application (a) secured by a program monitor (b).

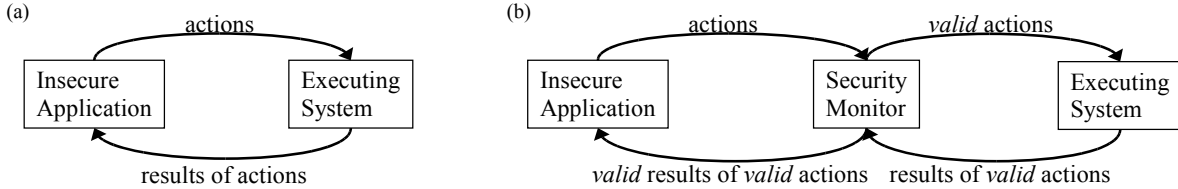


Figure 2: This paper's model of an insecure application with results (a) secured by a program monitor (b).

Although existing models of runtime mechanisms fit into the framework of Figure 1, that framework fails to capture the semantics of practical systems, in which actions return results and an application cannot continue executing after issuing an action a until receiving a result for a . Figure 2a depicts this more practical scenario, which encompasses application actions that:

- Return results, such as actions for dereferencing pointers, returning user input, reading and returning data in a file or network buffer, etc.
- Raise exceptions, such as actions for dereferencing (possibly null) pointers, writing to (possibly non-existent) files or network ports, etc. In this case, we simply treat the exceptions that can be raised as potential return values.
- Do not return results, such as actions for outputting text to a monitor or moving data from one register to another. In this case, the actions have an (implicit or explicit) `void` or `unit` return type, so we can view the underlying system as returning an actual `void` or `()` value upon completion of one of these actions.

Hence, all actions, even those not normally considered to return results, fit into a framework in which actions return results.

Besides interposing on and dynamically transforming actions, practical monitors can interpose on and dynamically transform action results, as Figure 2b illustrates. This is a crucial capability 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 email-client applications, or that applications cannot infer secret data based on the results they receive. Because existing frameworks for reasoning about generic runtime mechanisms do not model action results, one cannot use them to specify or reason about enforcing such result-sanitization policies.

Contributions This paper presents a new framework of runtime enforcement that captures the practical ability of monitors to transform both actions and results, as illustrated in Figure 2b. More specifically, we make the following contributions.

- Section 2 modifies existing definitions and notation for specifying systems and traces, to take into account the results of actions.

- Section 3 introduces *mandatory-results automata* (MRAs), models of runtime mechanisms that interpose on and enforce the security of two streams of events: application actions and the results of those actions. MRAs are obligated to output a result to the application for its most recently invoked action before the application can invoke another action.
- Section 4 defines policies, properties, and the precise circumstances under which MRAs enforce policies. Considering MRA-style enforcement forces us to deviate from existing definitions of policies and properties in some interesting ways.
- Section 5 analyzes the model of runtime monitoring created in the preceding sections to determine which policies MRAs can enforce. We find that MRAs enforce safety properties in highly uncertain environments. In more certain, but often realistic, environments, MRAs can enforce a rich set of policies that includes all elements of a new, succinctly specifiable set of policies we call the *action-life properties*. Besides the safety properties, action-life properties include some nonsafety liveness and nonsafety nonliveness properties.

This paper addresses one of what we consider to be the two principal shortcomings of existing runtime-enforcement frameworks: the inability to reason about (1) results of actions and (2) concurrency. We leave the complex issues of concurrency in monitoring frameworks for future work and focus here on monitoring *synchronous* actions (i.e., actions for which the application must receive a return value before continuing to execute) and the results of those actions.

2 Background Definitions and Notation

Before defining mandatory-results automata (MRAs) and what it means for an MRA to enforce a policy, we need to set up some basic definitions of, and notation for specifying, systems and traces. The definitions and notation presented in this section are extended versions of definitions and notation in previous work (extended to include results of actions) [4, 25, 23].

We define a system abstractly, in terms of the actions it can execute to perform computation and 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. A trace of a target application is the sequence of events that occur during a run of that application; the execution has finite length if the run terminates and infinite length otherwise. Because we assume synchronous actions, all executions contain alternating actions and results; that is, all complete executions under consideration in this paper must have one of the following forms, where r_0 is the result of action a_0 , r_1 is the result of a_1 , etc:

- \cdot (i.e., the execution has zero length and no events occur)
- $a_0; r_0; a_1; r_1; \dots; a_n; r_n$ (i.e., the execution terminates after receiving a result for a_n , which, for example, may have output a computed value or released a resource)
- $a_0; r_0; a_1; r_1; \dots; a_n$ (i.e., the execution terminates because the entity responsible for returning r_n never does so, which prevents the execution from continuing)
- $a_0; r_0; a_1; r_1; \dots$ (i.e., the execution diverges)

When event e occurs in execution x , we write $e \in x$. We denote the sequence of actions in an execution x as $actions(x)$, the sequence of results in x as $results(x)$, and the number of actions in finite-length x as $|x|$. Moreover, we denote the set of all finite-length executions on a system with event set E as E^* , all infinite-length executions as E^ω , and all finite- and infinite-length executions as E^∞ . We also let the metavariable e range over events, a over actions, r over results, ε and x over executions, and \mathcal{X} over sets of executions (i.e., subsets of E^∞). Sometimes it will be convenient to use α as a metavariable ranging over actions and \cdot , while ρ ranges over results and \cdot . Given a set of events E , A refers to all the actions in E , \dot{A} to $A \cup \{\cdot\}$, R to all the results in E , and \dot{R} to $R \cup \{\cdot\}$.

The notation $x;\varepsilon$ represents concatenation of two executions x and ε , the result of which must be a well-formed execution (i.e., x must be finite, and x cannot end with an action when ε is nonempty). When x is a finite prefix of ε we write $x \preceq \varepsilon$ or $\varepsilon \succeq x$ and say x *prefixes* ε or ε *extends* x . Finally, when E is clear from context, we make extensive use of abbreviations of the form $\exists x \preceq \varepsilon : F$ in place of $\exists x \in E^* : (x \preceq \varepsilon \wedge F)$, and $\forall x \preceq \varepsilon : F$ in place of $\forall x \in E^* : (x \preceq \varepsilon \implies F)$.

3 Mandatory-results Automata (MRAs)

Having defined systems and executions and set up some basic notation, we are ready to formally model monitors that behave as in Figure 2b. We call these models *mandatory-results automata* because they are security automata [25, 23] that must, because actions are synchronous, return a result to the target before seeing the next action the target wishes to execute. This mandatory-results constraint imposes a major limitation on MRA behaviors but accurately reflects the limitations of real program monitors (e.g., as created in Naccio [12], PoET [10], Polymer [5], LoPSiL [24], ConSpec [2], and many other enforcement systems).

3.1 Operational Semantics of MRAs

We model as MRAs monitors that can interpose on and transform synchronous actions and their results. An MRA M is tuple (E, Q, q_o, 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_o is M 's initial state, and δ is a (deterministic) transition function of the form $\delta : Q \times E \rightarrow Q \times E$, 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 [22, 23], 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 $\begin{smallmatrix} \alpha_i \\ \rho_o \end{smallmatrix} [q]_{\rho_i}^{\alpha_o}$ a *configuration* of MRA M , where q is M 's current state, α_i is either \cdot or the action currently 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). When we do not care about the values of α_i , α_o , ρ_i , and ρ_o in configuration $\begin{smallmatrix} \alpha_i \\ \rho_o \end{smallmatrix} [q]_{\rho_i}^{\alpha_o}$, we simply write the configuration as q . Also, we normally do not bother to write the dots in configurations, so $\begin{smallmatrix} a \\ \cdot \end{smallmatrix} [q]_r$ is the same as $\begin{smallmatrix} a \\ \cdot \end{smallmatrix} [q]_r$. The starting configuration of an MRA is $[q_o]$; that is, the monitor begins executing in its initial state with no events yet input or output. Notice that our bracket-based notation for configurations matches the graphic representation of monitors' inputs and outputs as depicted in Figure 2b.

We are now ready to describe the operational semantics of MRAs, as defined by a labeled single-step judgment whose form is $\begin{smallmatrix} \alpha_i \\ \rho_o \end{smallmatrix} [q]_{\rho_i}^{\alpha_o} \xrightarrow[\varepsilon_S]{\varepsilon_T} M \begin{smallmatrix} \alpha'_i \\ \rho'_o \end{smallmatrix} [q']_{\rho'_i}^{\alpha'_o}$. This judgment means that MRA M is taking a single

$$\boxed{\frac{\alpha_i [q]_{\rho_i}^{\alpha_o} \xrightarrow[\varepsilon_S]{\varepsilon_T} \alpha'_i [q']_{\rho'_i}^{\alpha'_o}}{}}$$

$$\frac{\text{next}_T = a}{\rho [q] \xrightarrow{a} \rho [q]} \text{ (Input-Act)}$$

$$\frac{\delta(q, a) = (q', a')}{\rho [q] \xrightarrow{a'} \rho [q']^{a'}} \text{ (Output-Act)}$$

$$\frac{\text{next}_S = r \quad \delta(q, r) = (q', -)}{\rho [q]^{a'} \xrightarrow{r} \rho [q']} \text{ (Input-Res)}$$

$$\frac{\delta(q, a) = (q', r)}{\rho [q] \xrightarrow{r} \rho [q']} \text{ (Output-Res)}$$

Figure 3: Single-step semantics of mandatory-results automata.

step from configuration $\rho_o [q]_{\rho_i}^{\alpha_o}$ to configuration $\rho'_o [q']_{\rho'_i}^{\alpha'_o}$ while building *target execution* ε_T and *system execution* ε_S . The target execution ε_T is the execution viewed by the target; in other words, ε_T is the sequence of actions that the target has sent to, and the results the target has received from, the monitor. Similarly, the system execution ε_S is the execution viewable by the system, that is, all the actions executed on, and the results of actions executed on, the underlying system. A final note about notation in our single-step judgment: we omit ε_T and ε_S from the judgment when they are empty, and because M will always be clear from the context, we normally omit it as well.

The definition of MRAs' single-step semantics appears in Figure 3. Four inference rules define all possible MRA transitions:

1. *Input-Act* enables the MRA to receive a new input action from the target (next_T is the next action generated by the target). The MRA can receive a new input action when it is in its initial configuration $[q_0]$ or when it is in a configuration of the form $\rho [q]$ (i.e., the MRA has most recently returned a result r to the target, so it is ready for another input action). No other transitions in Figure 3 take a step from configurations of the form $\rho [q]$, so if the target does not generate an action while the MRA is in such a configuration, the MRA will not transition from (and therefore terminates in) that configuration. We also note that *Input-Act* does not allow the MRA to change its state, because it is unnecessary to do so; the next transition after *Input-Act* must be either *Output-Act* or *Output-Res*, both of which allow the MRA to enter a new state based on the current input action.
2. *Output-Act* enables the MRA to delay returning a result for an input action a by first outputting an action a' .
3. *Input-Res* enables the MRA to update its state in response to receiving a result for its most recently output action a' (next_S is the next result generated by the system). A wildcard ($-$) exists in the premise of *Input-Res* in place of an event e because this rule, which causes no event to be output from the

MRA, ignores e 's value. No other transitions in Figure 3 take a step from configurations of the form ${}^a[q]^{a'}$, so if the executing system does not generate a result while the MRA is in such a configuration, the MRA will not transition from (and therefore terminates in) that configuration. Also, attentive readers may have noticed by now that MRAs transitioning according to the rules in Figure 3 only enter configurations with empty input results; this is because rule *Input-Res* implicitly merges two steps, the second of which always follows the first: ${}^a[q]^{a'} \xrightarrow{r} {}^a[q]_r \longrightarrow {}^a[q']$.

4. *Output-Res* enables the MRA to return a result to the target for the action most recently input to the MRA.

The simplicity of these inference rules, as much as is present, is the product of an incredible amount of effort and iteration through less-satisfactory definitions.

Several observations about the operational semantics of MRAs follow.

- An MRA can “accept” an input action by outputting it (with rule *Output-Act*), receiving and remembering its result r (with rule *Input-Res*), and then returning r to the application (with rule *Output-Res*).
- An MRA can “halt” an application by outputting an action like `exit`, if the underlying system can execute such an action. Alternatively, an MRA can enter an infinite loop in its transition function to block additional actions and results from being input and output.
- MRAs are indeed obligated to return results to applications before receiving new input actions. No transitions described above allow an MRA to input a new action until it has discharged the last input action by returning a result for it.
- 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., `getCurrentCloudCover`, `inputNextCharFromUser`, or more common actions like `readFileData` or `dereferenceLocation`) by having the system actually execute those actions. This design also implies that the *Input-Res* transition, and therefore the single-step relation for MRAs in general, may be nondeterministic.
- Just as MRAs are agnostic of how the executing system generates results, MRAs are also agnostic of whether and how the target generates actions. Agnosticism of target applications makes MRAs *purely dynamic* enforcement mechanisms; MRAs cannot use any knowledge of application source code (i.e., *how* applications generate actions) when enforcing properties. Because applications may generate actions nondeterministically, the *Input-Act* transition may also be nondeterministic.

These observations match our understanding of how real program monitors (such as can be specified in expressive aspect-oriented and policy-specification languages [16, 9, 24, 5]) behave.

Having defined the single-step semantics of MRAs, we define a multi-step judgment in the standard way as the reflexive, transitive closure of the single-step judgment. The multi-step judgment form is

${}_{\rho_o}^{\alpha_i} [q]_{\rho_i}^{\alpha_o} \xrightarrow[\varepsilon_S]{\varepsilon_T} {}_{\rho'_o}^{\alpha'_i} [q']_{\rho'_i}^{\alpha'_o}$, where ε_T and ε_S are the target and system executions formed (via concatenations)

during the multi-step transition from configuration ${}_{\rho_o}^{\alpha_i} [q]_{\rho_i}^{\alpha_o}$ to configuration ${}_{\rho'_o}^{\alpha'_i} [q']_{\rho'_i}^{\alpha'_o}$.

The multi-step judgment lets us reason about target and system executions after any finite number of transitions, but MRAs may make an infinite number of transitions while monitoring a program run. We therefore generalize the multi-step judgment to handle possibly infinite-length target and system executions built up through possibly infinitely many transitions. The judgment $\varepsilon_T \xleftrightarrow[M]{\leftarrow} \varepsilon_S$ will indicate that MRA M , beginning in its initial state, builds target and system executions ε_T and ε_S during its entire (possibly

infinite-transition) run. Intuitively, $\varepsilon_T \xleftrightarrow{M} \varepsilon_S$ holds if and only if, in any finite number of steps, (1) M builds target and system executions containing *every* prefix of ε_T and ε_S , and (2) conversely, M *only* builds target and system executions that are prefixes of ε_T and ε_S when receiving input events from ε_T and ε_S . Formally, we define $\varepsilon_T \xleftrightarrow{M} \varepsilon_S$ as follows.

Definition 1. For all MRA $M = (E, Q, q_0, \delta)$ and $\varepsilon_T, \varepsilon_S \in E^\infty$, $\varepsilon_T \xleftrightarrow{M} \varepsilon_S$ if and only if:

1. $\forall \varepsilon'_T \preceq \varepsilon_T : \forall \varepsilon'_S \preceq \varepsilon_S : \exists q \in Q : \exists x_T, x_S \in E^* : \left(q_0 \xrightarrow[x_S]{x_T} q \wedge \varepsilon'_T \preceq x_T \preceq \varepsilon_T \wedge \varepsilon'_S \preceq x_S \preceq \varepsilon_S \right)$
2. $\forall q \in Q : \forall x_T, x_S \in E^* : \left(q_0 \xrightarrow[x_S]{x_T} q \wedge \text{actions}(x_T) \preceq \text{actions}(\varepsilon_T) \wedge \text{results}(x_S) \preceq \text{results}(\varepsilon_S) \right) \implies (x_T \preceq \varepsilon_T \wedge x_S \preceq \varepsilon_S)$

Although the multi-step and $\varepsilon_T \xleftrightarrow{M} \varepsilon_S$ relations are nondeterministic for the same reasons the single-step relation is nondeterministic (i.e., targets and systems may generate actions and results nondeterministically), all three relations are deterministic for fixed sequences of input events. Lemmas 2 and 3 assure this determinism of the transition relations under fixed inputs.

Lemma 2. For all $M = (E, Q, q_0, \delta)$, $q, q' \in Q$, $\alpha_i, \alpha_o, \alpha'_i, \alpha'_o \in \dot{A}$, $\rho_i, \rho_o, \rho'_i, \rho'_o \in \dot{R}$, and $\varepsilon_T, \varepsilon_S, x_T, x_S \in E^*$:

$$\left(\begin{array}{l} \alpha_i [q]_{\rho_i}^{\alpha_o} \xrightarrow[\varepsilon_S]{\varepsilon_T} \alpha'_i [q']_{\rho'_i}^{\alpha'_o} \wedge \alpha_i [q]_{\rho_i}^{\alpha_o} \xrightarrow[x_S]{x_T} \alpha'_i [q']_{\rho'_i}^{\alpha'_o} \wedge \\ \text{actions}(\varepsilon_T) = \text{actions}(x_T) \wedge \text{results}(\varepsilon_S) = \text{results}(x_S) \end{array} \right) \implies (\varepsilon_T = x_T \wedge \varepsilon_S = x_S)$$

Lemma 3. For all MRAs $M = (E, Q, q_0, \delta)$ and $\varepsilon_T, \varepsilon_S, x_T, x_S \in E^\infty$: if $\varepsilon_T \xleftrightarrow{M} \varepsilon_S$, $x_T \xleftrightarrow{M} x_S$, $\text{actions}(\varepsilon_T) = \text{actions}(x_T)$, and $\text{results}(\varepsilon_S) = \text{results}(x_S)$ then $\varepsilon_T = x_T$ and $\varepsilon_S = x_S$.

All proofs appear in Appendix A.

3.2 Input and Output Executions

We adopt the notion that mechanisms enforce policies when they are *sound* and *transparent* [21, 9, 15, 23]. Soundness requires that the *observable* execution, which is the execution *output* from the mechanism, satisfies the desired policy. Soundness alone is unsatisfactory, though, for if enforcement only required soundness then a mechanism could enforce any property satisfied by execution x just by always outputting x (regardless of the input). Transparency prevents this situation by requiring mechanisms to preserve valid *input* executions; if the application and system provide valid input events to the mechanism, then the mechanism must output those valid inputs verbatim.

MRAs input actions from the target and results from the system, and output actions to the system and results to the target. Hence, sound MRAs always output valid sequences of actions (to the system) and results (to the target), while transparent MRAs never modify valid sequences of actions (from the target) and results (from the system). For example, consider an MRA $M = (E, \dot{R}, \cdot, \delta)$, where δ is defined as follows.

$$\delta(q, e) = \begin{cases} (\cdot, e) & \text{if } e \in A \text{ and } q = \cdot \\ (r, r) & \text{if } e \in R \text{ and } r = e - v \text{ (where } v \text{ is a secret value)} \\ (\cdot, q) & \text{if } e \in A \text{ and } q \neq \cdot \end{cases}$$

At a high level, when M receives an action a as input, it outputs a , receives a result r for a , and then outputs r stripped of any instances of secret-value v . M thus enforces a secret-value policy requiring v to never be returned to the target: M is sound because it never outputs a result containing v , and M is transparent because it outputs valid input events (i.e., actions and non- v -containing results) verbatim.

Reasoning about soundness and transparency requires reasoning about which executions enforcement mechanisms input and output. In the case of MRAs, understanding and defining precisely which executions

get input and output is surprisingly challenging, primarily for two reasons: (1) input and output executions may have infinite length, and (2) MRAs may input actions without inputting results for those actions, may input results for actions that were never input, may output actions without outputting results for those actions, and may output results for actions that were never output. This section overcomes these challenges to build a judgment of the form $\varepsilon_i \Downarrow_M \varepsilon_o$, which means that MRA M *transforms* possibly infinite-length input execution ε_i into possibly infinite-length output execution ε_o (the idea of viewing enforcement mechanisms as sound and transparent execution transformers comes from earlier work [22, 23]). Section 4 will use the $\varepsilon_i \Downarrow_M \varepsilon_o$ judgment to define enforcement formally.

3.2.1 Finite-length Input and Output Executions

We build up the \Downarrow_M judgment with two auxiliary definitions that show how to convert *finite-length* target and system executions (as constructed in the MRA multi-step relation) into *finite-length* input and output executions. That is, given finite-length target and system executions ε_T and ε_S , we define $input_*(\varepsilon_T, \varepsilon_S)$ to be the input execution used to build ε_T and ε_S , and $output_*(\varepsilon_T, \varepsilon_S)$ to be the output execution used to build ε_T and ε_S .

We reason that all actions in ε_T and results in ε_S must be part of $input_*(\varepsilon_T, \varepsilon_S)$, and all actions in ε_S and results in ε_T must be part of $output_*(\varepsilon_T, \varepsilon_S)$. More specifically, we build $input_*(\varepsilon_T, \varepsilon_S)$ by taking all the actions in ε_T and interspersing the results to those actions as first found in ε_S . Similarly, we build $output_*(\varepsilon_T, \varepsilon_S)$ by taking all the actions in ε_S and interspersing the results to those actions as first found in ε_T . Thus, we would convert target execution $a_1; r_1; a_2; r_2$ and system execution $a_1; r_2; a_2; r_1$ into input execution $a_1; r_2; a_2; r_1$ and output execution $a_1; r_1; a_2; r_2$. Similarly, we would convert target execution $a_1; r_1; a_2; r_2$ and system execution $a_2; r_1; a_1; r_2$ into input execution $a_1; r_2; a_2; r_1$ and output execution $a_2; r_2; a_1; r_1$.

However, issues arise when actions and results are not so well aligned. For example, if an action a in ε_T is not present in ε_S , then we have no result to include for a in the input execution! This situation can occur when the MRA inputs and directly returns a result for action a , without ever outputting a . In this case, we place a hole (\square) in the input execution to represent a 's nonexistent result (we assume $\square \notin E$ for all event sets E). This solution is reasonable because a must be part of the input execution but cannot logically be paired with any input result. Similarly, if a in ε_S is not present in ε_T (which occurs when the MRA outputs a without inputting a), then we place a \square in the output execution to represent a 's nonexistent result. And the same idea holds for unmatched results: if result r for action a appears in ε_S , but a is not in ε_T (which again occurs when the MRA outputs a without inputting a), then r is an input event for which no input action exists, so we place a \square in the input execution to represent r 's action. Finally, if result r for action a appears in ε_T , but a is not in ε_S (which occurs when the MRA returns result r for a to the target without outputting a), then r is an output result for which no output action exists, so we place a \square in the output execution to represent r 's nonexistent action.

The rules in Figure 4 formalize this process of converting finite-length target and system executions into finite-length, possibly \square -containing, input and output executions. Rule *Input-Basis* specifies that if the target and system executions are empty or contain a single action, then no results have been input, so the input execution should just be the target execution. Rule *Input-No-Act* specifies that if the system execution begins with an action a being output and a result r being input such that no corresponding action a was ever input, then the input execution should be $\square; r$ followed by whatever input execution gets built from the target execution and remaining system execution (excluding $a; r$). Rule *Input-No-Res* specifies that if the system execution begins with some action that also appears in the target execution (hence, rule *Input-No-Act* cannot apply) but the target execution begins with an action a' for which no result was ever input, then the input execution should be $a'; \square$ followed by whatever input execution gets built from the remaining target execution (excluding a' and its result) and the system execution (excluding any trailing action that was output for which no result got input). Finally, rule *Input-Act-and-Res* specifies that when the system execution begins with some action that also appears in the target execution (hence, rule *Input-No-Act* again cannot apply), and the target execution begins with an action a' that also appears the system execution, and the first time a' appears in the system execution it returns result r' , then the input execution should be $a'; r'$ followed by whatever input execution gets built from the non- a' -involving portions of the target and

$$\boxed{input_*(\varepsilon_T, \varepsilon_S) = \varepsilon_i}$$

$$\frac{}{input_*(\alpha_T, \alpha_S) = \alpha_T} \quad (Input-Basis)$$

$$\frac{a \notin \varepsilon_T}{input_*(\varepsilon_T, a; r; \varepsilon_S) = \square; r; input_*(\varepsilon_T, \varepsilon_S)} \quad (Input-No-Act)$$

$$\frac{\forall a \preceq \varepsilon_S : a \in \varepsilon_T \quad a' \notin \varepsilon_S}{input_*(a'; r; \varepsilon_T, \varepsilon_S; \alpha) = a'; \square; input_*(\varepsilon_T, \varepsilon_S)} \quad (Input-No-Res)$$

$$\frac{\forall a \preceq \varepsilon_S : a \in \varepsilon_T \quad a' \notin \varepsilon_S}{input_*(a'; \rho; \varepsilon_T, \varepsilon_S; a'; r'; \varepsilon'_S) = a'; r'; input_*(\varepsilon_T, \varepsilon_S; \varepsilon'_S)} \quad (Input-Act-and-Res)$$

$$\boxed{output_*(\varepsilon_T, \varepsilon_S) = \varepsilon_o}$$

$$\frac{}{output_*(\alpha_T, \alpha_S) = \alpha_S} \quad (Output-Basis)$$

$$\frac{a \notin \varepsilon_S}{output_*(a; r; \varepsilon_T, \varepsilon_S) = \square; r; output_*(\varepsilon_T, \varepsilon_S)} \quad (Output-No-Act)$$

$$\frac{\forall a \preceq \varepsilon_T : a \in \varepsilon_S \quad a' \notin \varepsilon_T}{output_*(\varepsilon_T; \alpha, a'; r; \varepsilon_S) = a'; \square; output_*(\varepsilon_T, \varepsilon_S)} \quad (Output-No-Res)$$

$$\frac{\forall a \preceq \varepsilon_T : a \in \varepsilon_S \quad a' \notin \varepsilon_T}{output_*(\varepsilon_T; a'; r'; \varepsilon'_T, a'; \rho; \varepsilon_S) = a'; r'; output_*(\varepsilon_T; \varepsilon'_T, \varepsilon_S)} \quad (Output-Act-and-Res)$$

Figure 4: Definitions of $input_*$ and $output_*$.

system executions. The rules defining the $output_*$ relation are analogous.

As a complex example of building input and output executions from target and system executions: given target execution $a_1; r_1; a_1; r_1; a_3; r_3; a_2; r_2$ and system execution $a_4; r_4; a_3; r_1; a_1; r_2$, the rules in Figure 4 build input execution $\square; r_4; a_1; r_2; a_1; \square; a_3; r_1; a_2; \square$ and output execution $a_4; \square; a_3; r_3; a_1; r_1; \square; r_1; \square; r_2$. Fortunately, our analyses rarely have to consider such complex examples; in the common case, $\varepsilon_T \preceq \varepsilon_S$ or $\varepsilon_S \preceq \varepsilon_T$ because transparent MRAs must ensure that the events they output eventually match the events they input, when those input events form a valid execution.

Now that we will sometimes consider executions with holes in them, we need to refine some of our definitions and notation from Section 2. First, $actions(\varepsilon)$ and $results(\varepsilon)$ will ignore \square s in ε (i.e., treat holes as dots). Similarly, we ignore trailing \square s in executions, so $a; r; a; \square$ is the same as $a; r; a$. Nontrailing holes in executions function as placeholders, though, and do count in our length operator (e.g., $|\square; r| = 2$). On a system with event set E , we will write E_\square for $E \cup \{\square\}$, making E_\square^* the set of all well-formed finite executions over E_\square , E_\square^ω the set of all well-formed infinite executions over E_\square , and $E_\square^\infty = E_\square^* \cup E_\square^\omega$. We call executions with no holes *natural* and executions with one or more holes *unnatural*. The only unnatural executions this paper considers are executions input to and output from MRAs; it never makes sense in our model for target or system executions to be unnatural.

As an MRA executes, every transition it makes appends some event onto the target or system execution. Extending the target or system execution by event e causes either (1) the input or output execution to be extended by e (e.g., $output_*(a, \cdot) = \square$ and $output_*(a; r, \cdot) = \square; r$), or (2) a hole in the input or output execution to get filled in by e (e.g., $output_*(a; r; a, \cdot) = \square; r$ and $output_*(a; r; a, a) = a; r$). For this reason, the notion of ε extending x when x is a prefix of ε , which works fine for target and system executions, does not always capture the semantics of extending input and output executions. Instead, we need a new notion of ε extending x when we can build ε by appending events to x and/or filling in holes in x . We write the formal judgment for this as $x \preceq_\square \varepsilon$, which holds on $x \in E_\square^*$ and $\varepsilon \in E_\square^\infty$ if and only if:

$$\exists x' \preceq \varepsilon : |x|=|x'| \wedge actions(x) \preceq actions(x') \wedge results(x) \preceq results(x') \quad (\text{EXTENSION WITH HOLES})$$

With these new definitions, we can show that $input_*$ and $output_*$ obey desirable properties:

Lemma 4. *For all event sets E and $\varepsilon_T, \varepsilon_S \in E^*$ and $\varepsilon_i, \varepsilon_o \in E_\square^*$, if $input_*(\varepsilon_T, \varepsilon_S) = \varepsilon_i$ and $output_*(\varepsilon_T, \varepsilon_S) = \varepsilon_o$ then $actions(\varepsilon_T) = actions(\varepsilon_i)$, $results(\varepsilon_S)$ is some permutation of $results(\varepsilon_i)$, $actions(\varepsilon_S) = actions(\varepsilon_o)$, and $results(\varepsilon_T)$ is some permutation of $results(\varepsilon_o)$.*

Lemma 5. *For all event sets E , target and system executions $\varepsilon_T, \varepsilon_S \in E^\infty$, actions $a \in A$, and results $r \in R$:*

- $input_*(\varepsilon_T, \varepsilon_S) \preceq_\square input_*(\varepsilon_T; a, \varepsilon_S)$
- $output_*(\varepsilon_T, \varepsilon_S) = output_*(\varepsilon_T; a, \varepsilon_S)$
- $input_*(\varepsilon_T, \varepsilon_S) = input_*(\varepsilon_T; r, \varepsilon_S)$
- $output_*(\varepsilon_T, \varepsilon_S) \preceq_\square output_*(\varepsilon_T; r, \varepsilon_S)$
- $input_*(\varepsilon_T, \varepsilon_S) = input_*(\varepsilon_T, \varepsilon_S; a)$
- $output_*(\varepsilon_T, \varepsilon_S) \preceq_\square output_*(\varepsilon_T, \varepsilon_S; a)$
- $input_*(\varepsilon_T, \varepsilon_S) \preceq_\square input_*(\varepsilon_T, \varepsilon_S; r)$
- $output_*(\varepsilon_T, \varepsilon_S) = output_*(\varepsilon_T, \varepsilon_S; r)$

Lemma 6. *For all event sets E and $\varepsilon \in E^*$, $input_*(\varepsilon, \varepsilon) = output_*(\varepsilon, \varepsilon) = \varepsilon$.*

Lemma 7. *For all event sets E , $\varepsilon \in E^*$, $a \in A$, and $r \in R$: $input_*(\varepsilon; a, \varepsilon) = \varepsilon; a$ and $output_*(\varepsilon; a, \varepsilon) = \varepsilon$ and $input_*(\varepsilon, \varepsilon; r) = \varepsilon; r$ and $output_*(\varepsilon, \varepsilon; r) = \varepsilon$.*

Lemma 8. *The $input_*$ and $output_*$ relations are total functions.*

3.2.2 Transformation of Possibly Infinite-length Executions

We generalize the $input_*$ and $output_*$ judgments, which operate on finite-length executions, into $input_\infty$ and $output_\infty$ judgments, which operate on possibly infinite-length executions, similar to the way we generalized the MRA multi-step judgment to the $\varepsilon_T \xleftrightarrow{M} \varepsilon_S$ judgment. Intuitively, $input_\infty(\varepsilon_T, \varepsilon_S) = \varepsilon_i$ when (1) every prefix of ε_i eventually gets input from target execution ε_T and system execution ε_S , and (2) only prefixes of ε_i get input from ε_T and ε_S , though the prefixes of ε_i that get input may temporarily contain extra \square s that ε_i lacks.

Definition 9. For all event sets E and $\varepsilon_T, \varepsilon_S \in E^\infty$ and $\varepsilon_i \in E_\square^\infty$, $input_\infty(\varepsilon_T, \varepsilon_S) = \varepsilon_i$ if and only if:

1. $\forall x_i \preceq \varepsilon_i : \exists x_T \preceq \varepsilon_T : \exists x_S \preceq \varepsilon_S : x_i \preceq input_*(x_T, x_S)$
2. $\forall x_T \preceq \varepsilon_T : \forall x_S \preceq \varepsilon_S : input_*(x_T, x_S) \preceq_\square \varepsilon_i$

The $output_\infty$ judgment is defined similarly.

Definition 10. For all event sets E and $\varepsilon_T, \varepsilon_S \in E^\infty$ and $\varepsilon_o \in E_\square^\infty$, $output_\infty(\varepsilon_T, \varepsilon_S) = \varepsilon_o$ if and only if:

1. $\forall x_o \preceq \varepsilon_o : \exists x_T \preceq \varepsilon_T : \exists x_S \preceq \varepsilon_S : x_o \preceq output_*(x_T, x_S)$
2. $\forall x_T \preceq \varepsilon_T : \forall x_S \preceq \varepsilon_S : output_*(x_T, x_S) \preceq_\square \varepsilon_o$

Lemma 11. The $input_\infty$ and $output_\infty$ relations are total functions.

At last, we can define the transformation of a possibly infinite-length input execution ε_i into a possibly infinite-length output execution ε_o by an MRA M , written as $\varepsilon_i \Downarrow_M \varepsilon_o$. This judgment holds when M builds (possibly infinite-length) target and system executions ε_T and ε_S such that $input_\infty(\varepsilon_T, \varepsilon_S) = \varepsilon_i$ and $output_\infty(\varepsilon_T, \varepsilon_S) = \varepsilon_o$.

Definition 12. For all MRAs $M=(E, Q, q_0, \delta)$ and executions $\varepsilon_i, \varepsilon_o \in E^\infty$: $\varepsilon_i \Downarrow_M \varepsilon_o$ if and only if $\exists \varepsilon_T, \varepsilon_S \in E^\infty : (\varepsilon_T \xleftrightarrow{M} \varepsilon_S \wedge input_\infty(\varepsilon_T, \varepsilon_S) = \varepsilon_i \wedge output_\infty(\varepsilon_T, \varepsilon_S) = \varepsilon_o)$.

Because MRA transition relations are deterministic on fixed inputs (by Lemma 3), we intuitively expect that if MRA M transforms ε_i into ε_o , then M always transforms ε_i into ε_o . The following theorem shows that our definitions respect this intuition and the \Downarrow_M relation is a function from input to output executions—but \Downarrow_M is only a partial function because M may never build certain input executions.

Theorem 13. For all MRAs $M = (E, Q, q_0, \delta)$ and $\varepsilon_i, \varepsilon_o, x_o \in E_\square^\infty$, if $\varepsilon_i \Downarrow_M \varepsilon_o$ and $\varepsilon_i \Downarrow_M x_o$ then $\varepsilon_o = x_o$.

4 MRA-based Enforcement

Having defined MRAs' operational semantics and input and output executions, we can define policies, properties, and what it means for an MRA to enforce a policy (in terms of soundness and transparency).

4.1 Policies and Properties

A *policy* (or equivalently, a hyperproperty [7]) is a predicate on sets of executions [25]. A set of executions $\mathcal{X} \subseteq E_\square^\infty$ satisfies a policy P if and only if $P(\mathcal{X})$. Some policies are also *properties*. Policy P is a property if and only if there exists a *characteristic predicate* \hat{P} over E_\square^∞ such that for all $\mathcal{X} \subseteq E_\square^\infty$, the following is true [25]:

$$P(\mathcal{X}) \iff \forall x \in \mathcal{X} : \hat{P}(x) \quad (\text{PROPERTY})$$

There is a one-to-one correspondence between a property P and its characteristic predicate \hat{P} , so we use the notation \hat{P} unambiguously to refer to both a characteristic predicate and the property it induces.

The distinction between properties and more general policies is important when reasoning about runtime enforcement because runtime mechanisms monitor a single execution at a time and make decisions about whether that single execution is secure, thereby enforcing properties rather than policies. In contrast, static-analysis mechanisms can enforce nonproperty policies by considering multiple executions of an application.

Our definitions of policies and properties deviate from existing definitions in that policies here are predicates over subsets of E_{\square}^{∞} rather than E^{∞} . That is, our policies must reason about holes in executions. In general, policies must specify which observable executions are valid, but in systems with MRAs, observable executions may contain \square s, so policies in systems with MRAs have to consider \square -containing executions. However, considering \square s in output executions gives policies great flexibility in defining valid behaviors: an obligation policy might require action a to appear and would therefore consider execution $\square; r$ invalid, while an access-control policy might disallow a actions and consider $\square; r$ a perfectly valid execution.

Also, because our definitions of policies and properties operate on executions containing results, they may take results into consideration. For example, a policy might be satisfied by exactly those sets of executions \mathcal{X} in which the results of all actions in all executions in \mathcal{X} form a complete set of natural numbers. This policy is not a property because there is no predicate \hat{P} that can look at individual executions in isolation to determine whether the results of all actions in all executions of \mathcal{X} form a complete set of natural numbers. On the other hand, consider a policy satisfied by exactly those sets of executions \mathcal{X} in which no result of any action (in any execution in \mathcal{X}) contains a secret value v . This policy is a property because it is satisfied if and only if every execution in \mathcal{X} has exactly zero v -containing results. An MRA can enforce this secret-value property, as described in Section 3.2.

Given a property \hat{P} and a natural, finite execution ε , we will often find it useful to know whether ε can be extended into a valid execution. When ε can be made valid by extending it with any natural sequence of events, we say ε is *alive*; when ε can be made valid by extending it with a function-generated sequence $a_0; \square; a_1; \square; a_2; \square; \dots$, we say ε is *action-alive*, or *aalive*. Formally, on a system with event set E , $alive_{\hat{P}}(\varepsilon)$ is true for $\varepsilon \in E^*$ if and only if:

$$\exists x \in E^{\infty} : \left(\hat{P}(\varepsilon; x) \right) \quad (\text{ALIVE})$$

and $aalive_{\hat{P}, f}(\varepsilon)$, which requires $f : E^* \times \mathbb{N} \rightarrow \dot{A}$, is true for $\varepsilon \in E^*$ if and only if:

$$\exists x \in E_{\square}^{\infty} : \left(\hat{P}(\varepsilon; x) \wedge results(x) = \cdot \wedge actions(x) = f(\varepsilon, 0); f(\varepsilon, 1); \dots \right) \quad (\text{ACTION-ALIVE})$$

Because \hat{P} will always be clear from the context, we will omit it as a subscript in future $alive(\varepsilon)$ and $aalive_f(\varepsilon)$ 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 analyses of security automata, this paper limits its scope to properties with predicates $\hat{P}(\varepsilon)$ and $alive(\varepsilon)$ that are decidable over finite ε .

Properties, such as the secret-value policy described above, which specify that “nothing bad ever occurs” are called *safety* properties [18]. This well-studied class of properties includes commonly enforced properties such as access-control properties (specifying that no invalid accessing of resources ever occurs). Technically, safety means that every invalid execution must have some invalid prefix that is *dead* (i.e., not alive) [3], so a property \hat{P} on a system with event set E is a safety property if and only if:

$$\forall \varepsilon \in E^{\infty} : \left(\neg \hat{P}(\varepsilon) \implies \exists x \preceq \varepsilon : \neg alive(x) \right) \quad (\text{SAFETY})$$

In contrast, *liveness* properties are ones in which all finite executions are alive [4]. Like safety properties, liveness properties are well studied, but unlike safety properties, liveness includes particularly difficult- or impossible-to-enforce properties such as that an application will *eventually* input a particular value or terminate. Formally, a property \hat{P} on a system with event set E is a liveness property if and only if:

$$\forall \varepsilon \in E^* : alive(x) \quad (\text{LIVENESS})$$

4.2 Enforcement

We can now formalize *enforcement* in terms of (1) soundness: if an MRA M ever outputs (possibly unnatural) ε_o then ε_o must satisfy the desired property, and (2) transparency: M must input, and output verbatim, all valid (and natural) sequences of events. The notation and definitions built up to this point enable the following elegant formalization of enforcement.

Definition 14. *MRA M enforces property \hat{P} on a system with event set E if and only if:*

1. $\forall \varepsilon_i, \varepsilon_o \in E_{\square}^{\infty} : (\varepsilon_i \Downarrow_M \varepsilon_o) \implies \hat{P}(\varepsilon_o)$ (*Soundness*)
2. $\forall \varepsilon \in E^{\infty} : \hat{P}(\varepsilon) \implies (\varepsilon \Downarrow_M \varepsilon)$ (*Transparency*)

5 Analysis of MRA-enforceable Policies

MRAs as defined so far operate in a highly uncertain environment, completely unaware of future behaviors of the target and executing system. Often in practice, though, runtime monitors operate in environments that provide slightly more certainty about target and system behaviors. This section analyzes the properties MRAs can enforce in both of these environments.

5.1 MRA-enforceable Policies in Highly Uncertain Environments

MRAs as defined so far never know whether they will receive additional input; target applications and executing systems may cease to generate actions and results at any time. This uncertainty implies that MRAs can only ever output events that are known to be valid. An MRA M cannot output even a temporarily invalid event e because after doing so, M may receive no additional inputs and therefore have no additional opportunities to extend the currently invalid output, to make it valid. Hence, M must give up and halt its output as soon as it inputs any invalid event. Giving up in this situation prevents M from being transparent on any valid extensions of its current input (extending beyond the invalid event), so M can only enforce safety properties (all extensions of invalid executions must be invalid, and all invalid infinite executions x must become invalid at some finite point to prevent M from outputting x).

On the other hand, MRAs can enforce *all* safety properties that consider \cdot valid. MRAs do so by outputting all valid input events verbatim and halting¹ as soon as an invalid input event is detected. In this way, the MRA will only output valid executions, and if the input execution is valid then the MRA's output will match its input (because the property is a safety property, which implies that all prefixes of valid natural executions must also be valid).

Theorem 15. *A property \hat{P} on a system with event set E can be enforced by some MRA M if and only if:*

1. $\hat{P}(\cdot)$ (*Empty-validity*)
2. $\forall \varepsilon \in E^{\infty} : (\neg \hat{P}(\varepsilon) \implies \exists x \preceq \varepsilon : \neg \text{alive}(x))$ (*Safety*)

Please see Appendix A for the theorem proofs.

5.2 MRA-enforceable Policies in More Certain Environments

In practice, enforcement systems often provide a slightly more certain monitoring environment than that of our current model. The increased certainty comes from the following two guarantees.

¹In our proof of Theorem 15, the MRA “halts” by entering an infinite loop to block additional inputs and outputs. MRAs in practice could enter infinite loops but would typically halt execution more straightforwardly by invoking an `exit` action. We use the infinite-loop approach for Theorem 15 because it does not limit us to systems with `exit`-style actions.

1. Target applications must terminate with an explicit `exit` action. Real enforcement systems like Nacchio [12], PoET [11, 9], and Polymer [5], which rewrite targets to inline monitoring code, guarantee that targets terminate by executing monitor-observable `exit` actions.
2. Executing systems must eventually return some result for every action they are asked to execute. System users typically expect machine-code instructions, bytecode instructions, system calls in well-established operating systems, and API-methods in well-established libraries to terminate reliably (possibly with an abnormal return value, like an exception).

There are a several technicalities to address before analyzing the policies MRAs enforce in these more certain environments. First, we will write MRA_{exit} to mean an MRA operating in an environment in which terminating applications are obligated to make `exit` the last action they generate. For generality, and to model practical enforcement systems, our MRA_{exit} are *not* obligated to output `exit` actions before terminating. However, we can constrain MRA_{exit} to always terminate with an `exit` action by requiring them to enforce policies in which all valid finite executions end with `exit`. We also assume that no result ever gets returned (to targets or MRAs) for `exit` actions.

MRA_{exit} in our model input *all* application actions, which gives them knowledge that, after outputting a result, they will receive another input action and get a chance to make additional output. However, for performance reasons real monitors typically only input a small security-relevant subset of target actions, which prevents them from knowing that they will receive an additional input action after outputting a result—the target may go on infinitely generating only non-security-relevant actions, imperceptible to the monitor. Practical monitors (such as those in the Polymer policy library [5]) work around this problem by considering clock ticks to be security relevant; that is, practical monitors set up timers to interrupt and transfer control to the monitor after a finite amount of time, so the monitors (on systems with `exit` actions) know, after outputting a result, that they will receive an additional input action. Hence, MRA_{exit} model the powers of monitors in real enforcement systems.

We will also write MRA_{\leftarrow} to mean an MRA that is guaranteed to receive a result for every action it outputs to the executing system. We assume that whenever an MRA_{\leftarrow} steps with rule *Output-Act* (of Figure 3), it will always next step with rule *Input-Res*. When we want to refer to MRAs operating in environments with both `exit` actions and guaranteed results for executed actions, we will call those automata $MRA_{exit, \leftarrow}$.

5.2.1 Analysis of MRA_{exit} and MRA_{\leftarrow}

MRA_{exit} and MRA_{\leftarrow} enforce different, but similar, sets of policies. The similarity in the enforcement powers of MRA_{exit} and MRA_{\leftarrow} stems from the symmetry between (1) MRA_{exit} knowing that an action will be input after every result is output and (2) MRA_{\leftarrow} knowing that a result will be input after every action is output.

The following two theorems highlight this symmetry. MRA_{exit} and MRA_{\leftarrow} both basically enforce safety properties, except that invalid finite execution x may be alive if there exists an event e such that $x;e$ is valid. For example, an MRA_{exit} can enforce a nonsafety property that only considers executions a and $a;r;exit$ valid. We call this an *action-escape-safety* property because invalid finite executions (e.g., \cdot and $a;r$) may “escape death” by appending some action (e.g., \cdot escapes death by appending a to make a valid a execution, and $a;r$ escapes death by appending `exit` to make a valid $a;r;exit$ execution). An MRA_{exit} can enforce this property because it knows (due to the presence of `exit` actions) that it will always have an opportunity to output an initial a action, as well as an `exit` action after outputting $a;r$.

Theorem 16. *A property \hat{P} on a system with event set E can be enforced by some MRA_{exit} M if and only if \hat{P} is satisfiable and:*

$$\forall \varepsilon \in E^\infty : \neg \hat{P}(\varepsilon) \implies \left(\left(\exists x \preceq \varepsilon : \neg \text{alive}(x) \right) \vee \left(\exists a \in A : \hat{P}(\varepsilon; a) \right) \right) \quad (\text{ACTION-ESCAPE SAFETY})$$

MRA_{\leftarrow} analogously enforce *result-escape-safety* properties.

Theorem 17. *A property \hat{P} on a system with event set E can be enforced by some $MRA_{\leftarrow} M$ if and only if $\hat{P}(\cdot)$ and:*

$$\forall \varepsilon \in E^\infty : \neg \hat{P}(\varepsilon) \implies \left(\left(\exists x \preceq \varepsilon : \neg \text{alive}(x) \right) \vee \left(\exists r \in R : \hat{P}(\varepsilon; r) \right) \right) \quad (\text{RESULT-ESCAPE SAFETY})$$

5.2.2 Analysis of $MRA_{\text{exit}, \leftarrow}$

By themselves, MRA_{exit} and MRA_{\leftarrow} enforce relatively little more than safety properties. Combining **exit** actions with guaranteed results for actions, though, enables $MRA_{\text{exit}, \leftarrow}$ to enforce a remarkably larger set of properties. The remainder of this section develops bounds on the properties $MRA_{\text{exit}, \leftarrow}$ can enforce.

Lower Bounds on $MRA_{\text{exit}, \leftarrow}$ -enforceable Properties $MRA_{\text{exit}, \leftarrow}$ can enforce any *action-life* property. Action-life properties are those that satisfy the following constraints.

1. The empty execution must be action-alive, so an $MRA_{\text{exit}, \leftarrow}$ can output a valid execution $a_0; \square; a_1; \square; \dots$ even when its only input is an **exit** action. Formally, there must exist a decidable $f : E^* \times \mathbb{N} \rightarrow \dot{A}$ such that $\text{alive}_f(\cdot)$.
2. All alive executions ε either must already be action-alive or there must be some computable result r such that $\varepsilon; r$ is action-alive. We call this the *implied-aalive* constraint because it basically states that an alive execution implies an aalive execution. An $MRA_{\text{exit}, \leftarrow}$ can enforce implied-aalive properties because it can output all alive prefixes of its input with assurance that if an **exit** action gets input then it can make its output valid by extending the current output with a sequence of actions. Formally, implied-aalive properties require that, using the same function f from above, there exists another decidable function $g : E^* \rightarrow \dot{R}$ such that $\forall \varepsilon \in E^* : \left(\text{alive}(\varepsilon) \implies \text{aalive}_f(\varepsilon; g(\varepsilon)) \right)$.
3. Finally, action-life properties must be ω -*safety properties*, which are properties for which the normal safety constraint only has to hold on infinite-length executions. Invalid *finite*-length executions may not have a dead prefix (and may therefore be alive), but invalid *infinite*-length executions must have a dead prefix. An $MRA_{\text{exit}, \leftarrow}$ can enforce ω -safety properties because it can output all alive prefixes of an invalid infinite execution x without risking outputting the entire x . Formally, ω -safety requires $\forall \varepsilon \in E^\omega : \left(\neg \hat{P}(\varepsilon) \implies \exists x \preceq \varepsilon : \neg \text{alive}(x) \right)$.

The following theorem states that action-life properties are a lower bound on what $MRA_{\text{exit}, \leftarrow}$ can enforce. The theorem's proof (in Appendix A) shows how to construct an $MRA_{\text{exit}, \leftarrow}$ to enforce any given action-life property.

Theorem 18. *A property \hat{P} on a system with event set E can be enforced by some $MRA_{\text{exit}, \leftarrow} M$ if there exist decidable $f : E^* \times \mathbb{N} \rightarrow \dot{A}$ and $g : E^* \rightarrow \dot{R}$ such that:*

1. $\text{aalive}_f(\cdot)$ (*Empty-aalive*)
2. $\forall \varepsilon \in E^* : \left(\text{alive}(\varepsilon) \implies \text{aalive}_f(\varepsilon; g(\varepsilon)) \right)$ (*Implied-aalive*)
3. $\forall \varepsilon \in E^\omega : \left(\neg \hat{P}(\varepsilon) \implies \exists x \preceq \varepsilon : \neg \text{alive}(x) \right)$ (ω -*safety*)

All safety properties satisfied by \cdot are action-life properties (because with safety properties, alive executions must be valid and therefore action-alive). Also, some interesting nonsafety properties, including some liveness and some nonliveness properties, are action-life properties. A simple property that is both action-life and liveness is nontermination, which considers all infinite, and only infinite, executions valid, making all finite executions alive and action-alive. An $MRA_{\text{exit}, \leftarrow} M$ can enforce nontermination simply by outputting all input events verbatim until an **exit** action is input, at which point M enters a loop to output some infinite sequence of actions.

Other sorts of nonsafety action-life properties include bounded action-obligation properties, which consider some input event invalid unless particular actions are later output (within a bounded number of steps). For example, a bounded resource-availability property may specify that a target may acquire and use any resource r only if, after using r for the first time, it makes one of its next two actions release r . In this case, the target is obligated to release r within two actions of its first use. This is a nonsafety property because an execution ending with the first use of r is invalid but can be extended into a valid execution by appending an action to release r . It is also a nonliveness property because some finite executions, like $\text{acquire}(r); _ ; \text{use}(r); _ ; \text{use}(r); _ ; \text{use}(r)$, are dead. And this is an action-life property because (1) the empty execution is valid (and therefore action-alive), (2) all alive finite executions are either valid (and therefore action-alive) or can be made valid by appending resource-release actions, and (3) for an infinite execution to be invalid, there must be some prefix that is dead because that prefix contains a resource use without a release within two actions. Note that *unbounded* obligation properties, such as unbounded resource-availability, are not action-life properties because they can violate ω -safety. For example, without the two-action bound in the previous example, all prefixes of an invalid infinite execution can be alive (because every prefix could be made valid by appending some number of resource-releasing actions). No $\text{MRA}_{\text{exit}, \leftarrow}$ can enforce such unbounded obligation properties because the MRA would be unable to decide when to output the obligatory events (e.g., resource-releasing actions) without violating transparency in the future, if the MRA later receives those obligatory events as input. For the same reason, $\text{MRAs}_{\text{exit}, \leftarrow}$ in general cannot enforce properties requiring that some sequence of events *eventually* (without a bound) appears in an execution.

As discussed in Section 1, some interesting policies constrain the results that can be returned to target applications, and one of the contributions of this paper is a framework for reasoning about such policies. Nonetheless, security policies often ignore results, only caring about which actions get executed. For example, the bounded resource-availability property of the previous paragraph ignores results. As another example, a property may specify that only certain users may execute certain actions, and a user application is allowed to see a result for an action if and only if the application is allowed to execute that action in the first place. It is interesting to consider these common sorts of *results-ignoring properties*, which can decide the validity of an execution only by examining its actions. A property \hat{P} is a results-ignoring property on a system with event set E if and only if:

$$\forall \varepsilon, x \in E_{\square}^{\infty} : (\text{actions}(\varepsilon) = \text{actions}(x)) \implies (\hat{P}(\varepsilon) \iff \hat{P}(x)) \quad (\text{RESULTS-IGNORING PROPERTY})$$

Consider a results-ignoring property \hat{P} such that there exists a function f that can input any finite alive execution ε and output, one action at a time, a valid extension of ε . Then this \hat{P} is an implied-alive property because every alive execution must be action-alive (given that \hat{P} ignores results and function f exists). We also observe that for all satisfiable results-ignoring properties, the empty execution \cdot must be action-alive. Combining these observations with Theorem 18, we obtain the following nice corollary, which states that $\text{MRAs}_{\text{exit}, \leftarrow}$ are powerful enforcers of results-ignoring properties.

Corollary 19. *For all satisfiable, ω -safety, results-ignoring properties \hat{P} , if there exists a decidable function that, when given any alive execution ε , can produce (action by action) a valid extension of ε , then there exists an $\text{MRA}_{\text{exit}, \leftarrow}$ that enforces \hat{P} .*

Properties satisfying this corollary's constraints include action-only transactional properties, which specify that executions are valid if and only if they are comprised of a (possibly infinite) sequence of valid action-based transactions. For example, a property may specify that valid executions are exactly those that match either the pattern $((a_1; _ ; a_2; _)|(a_3; _ ; a_4; _ ; a_2; _))^{\omega}$, where wildcards ($_$) can be any results (including \square), or the pattern $((a_1; _ ; a_2; _)|(a_3; _ ; a_4; _ ; a_2; _))^* \text{exit}$. Here, the valid transactions are $a_1; _ ; a_2; _$ and $a_3; _ ; a_4; _ ; a_2; _$. This action-only transactional property is satisfiable, results-ignoring, and ω -safety, so by Corollary 19 it is enforceable by $\text{MRAs}_{\text{exit}, \leftarrow}$ (using the enforcement technique described in Theorem 18's proof). On the other hand, no $\text{MRA}_{\text{exit}, \leftarrow} M$ can enforce general transactional properties, such as one in which the valid transactions are $a_1; r_1; a_2; r_2$ and $a_1; r_3; a_2; r_2$, because when processing a transaction, M

would have to output a_1 to be transparent on a_1 's result (which may be either r_1 or r_3), but if the next action input after M outputs r_1 or r_3 is **exit** then M cannot output the final r_2 that is needed to complete the current transaction.

Upper Bounds on $\text{MRA}_{\text{exit},\leftarrow}$ -enforceable Properties Theorem 18's lower bound on properties enforceable by $\text{MRAs}_{\text{exit},\leftarrow}$ is also an upper bound if we constrain $\text{MRAs}_{\text{exit},\leftarrow}$ to output all alive input executions verbatim (that is, for all $\text{MRAs}_{\text{exit},\leftarrow} M$ and alive $\varepsilon \in E^* : \varepsilon \Downarrow_M \varepsilon$). An $\text{MRA}_{\text{exit},\leftarrow} M$ can only enforce an action-life property \hat{P} under this constraint because (1) the empty execution must be action-alive in case M 's only input is an **exit** action; (2) \hat{P} must be an implied-aalive property in case M inputs **exit** after inputting (and therefore outputting) an alive execution; and (3) \hat{P} must be an ω -safety property to prevent M from outputting all alive prefixes of an invalid, infinite-length input execution.

Theorem 20. *If a property \hat{P} on a system with event set E can be enforced by some $\text{MRA}_{\text{exit},\leftarrow} M$ such that $\forall \varepsilon \in E^* : (\text{alive}(\varepsilon) \implies \varepsilon \Downarrow_M \varepsilon)$, then there exist decidable $f : E^* \times \mathbb{N} \rightarrow \dot{A}$ and $g : E^* \rightarrow \dot{R}$ such that:*

1. $\text{aalive}_f(\cdot)$ (Empty-aalive)
2. $\forall \varepsilon \in E^* : (\text{alive}(\varepsilon) \implies \text{aalive}_f(\varepsilon; g(\varepsilon)))$ (Implied-aalive)
3. $\forall \varepsilon \in E^\omega : (\neg \hat{P}(\varepsilon) \implies \exists x \preceq \varepsilon : \neg \text{alive}(x))$ (ω -safety)

Theorem 20 is useful because its constraint that $\text{MRAs}_{\text{exit},\leftarrow}$ output all alive inputs verbatim is not a big departure from reality. In fact, for all $\text{MRAs}_{\text{exit},\leftarrow} M$, alive executions ε , and arbitrary executions x , it is the case that $\varepsilon \Downarrow_M x$ implies $\varepsilon = x$. This is because M must be transparent on some valid input ε' that extends ε , so M must eventually input all events in ε , and output those events in the same order, when its input turns out to be ε' . Thus, given that M builds up exactly the natural input execution ε and that M must output one event for every event input (including the last input event of ε , to ensure that M can receive the next input of ε'), M must produce ε as its output execution when producing ε as its input execution. Combining Theorems 18 and 20, then, we see that action-life properties serve as a good approximation—and the best approximation we know of—for the exact set of (i.e., tight bounds on) properties that $\text{MRAs}_{\text{exit},\leftarrow}$ can enforce.

However, an $\text{MRA}_{\text{exit},\leftarrow} M$ may sometimes not build input executions matching all alive $\varepsilon \in E^*$; recall that \Downarrow_M is only a partial function. We therefore cannot say $\varepsilon \Downarrow_M \varepsilon$ for all alive executions ε . For example, consider M enforcing a property \hat{P} that specifies only executions **exit** and $a; r; \text{exit}$ as valid. M may enforce this \hat{P} with the following (high-level) transition function: if the first action input is not a then output **exit**; otherwise output r , then input the next action, then output a , then input some result for a , and finally output **exit**. This transition function guarantees that all output executions are either **exit** or $a; r; \text{exit}$ (soundness), and that **exit** \Downarrow_M **exit** and $a; r; \text{exit}$ \Downarrow_M $a; r; \text{exit}$ (transparency). At the same time, it is *not* the case that $a; r \Downarrow_M a; r$, despite the fact that $a; r$ is alive, because M can only input r after it has input **exit** (so M builds input executions $a, a; \square; \text{exit}$ and $a; r; \text{exit}$, but never just $a; r$). M can postpone outputting a in this example because it knows that r is the only result for a that could possibly make the current input execution valid; the postponement of outputting a would not be possible if another execution $a; r'; \text{exit}$ were also valid.

Let us examine which of the action-life constraints $\text{MRAs}_{\text{exit},\leftarrow}$ can violate when we lift the requirement of Theorem 20 that they have to output all alive inputs verbatim. First, we find that whenever a property \hat{P} is enforceable by an $\text{MRA}_{\text{exit},\leftarrow}$, the empty execution must indeed be action-alive with respect to \hat{P} .

Theorem 21. *If a property \hat{P} on a system with event set E can be enforced by some $\text{MRA}_{\text{exit},\leftarrow} M$ then there exists decidable $f : E^* \times \mathbb{N} \rightarrow \dot{A}$ such that $\text{aalive}_f(\cdot)$.*

Second, $\text{MRAs}_{\text{exit},\leftarrow}$ can enforce non-implied-aalive properties. An $\text{MRA}_{\text{exit},\leftarrow}$ can do so by postponing outputting its action inputs, outputting valid results in the meantime, until it is assured that its action inputs are valid when combined with the results it has been outputting. The proof of Theorem 22 constructs a concrete example of such an $\text{MRA}_{\text{exit},\leftarrow}$ enforcing a non-implied-aalive property.

Theorem 22. *There exist $MRA_{exit, \leftrightarrow} M$ and property \hat{P} on a system with event set E such that M enforces \hat{P} but \hat{P} is not an implied-alive property—that is, $\exists \varepsilon \in E^* : \left(\text{alive}(\varepsilon) \wedge \neg \text{aalive}_f(\varepsilon; g(\varepsilon)) \right)$, for all decidable $f : E^* \times \mathbb{N} \rightarrow \dot{A}$ and $g : E^* \rightarrow \dot{R}$.*

Third, $MRA_{S_{exit, \leftrightarrow}}$ can enforce non- ω -safety properties.

Theorem 23. *There exist $MRA_{exit, \leftrightarrow} M$ and property \hat{P} on a system with event set E such that M enforces \hat{P} but \hat{P} is not an ω -safety property—that is, $\exists \varepsilon \in E^\omega : \left(\neg \hat{P}(\varepsilon) \wedge \forall x \preceq \varepsilon : \text{alive}(x) \right)$.*

We saw earlier that $MRA_{S_{exit, \leftrightarrow}}$ can enforce nontermination. Now, Theorem 23’s proof constructs an $MRA_{exit, \leftrightarrow}$ that enforces a kind of termination property \hat{P} . This \hat{P} requires repetitions of $a; r$ to be finite, though repetitions of $\square; r$ may be infinite; technically, $\hat{P}(x)$ holds for $x \in E_{\square}^\infty$ if and only if x matches one of the patterns $(a; r)^* \text{exit}$ or $(\square; r)^* \text{exit}$ or $(\square; r)^\omega$. Property \hat{P} is an *action-termination property* because executions with actions are valid if and only if they contain a finite sequence of a actions. \hat{P} is not an ω -safety property because infinite execution $a; r; a; r; \dots$ is invalid despite all of its prefixes being alive. Although $MRA_{S_{exit, \leftrightarrow}}$ can enforce only very limited kinds of termination properties like this \hat{P} —and $MRA_{S_{exit, \leftrightarrow}}$ certainly cannot enforce general termination, which requires *all* executions to terminate—we are impressed that it is possible to (soundly and transparently) enforce some kind of termination property with a realistic mechanism.

6 Related Work

The past decade has seen many efforts to model runtime monitors as security automata and analyze their enforcement powers; a recent article surveys the results [23]. To briefly summarize previous work: Schneider showed that truncation automata (which accept valid application actions and halt target applications upon inputting invalid actions) enforce only safety properties [25]; Viswanathan, Kim, and others refined these safety bounds by adding computability constraints to the safety properties being enforced [27, 17]; Fong showed that truncation automata with limited memory (e.g., with space to store only a constant number of actions input) can still enforce practically useful safety policies [14]; Walker and Aktug et al. presented techniques for statically guaranteeing that policies represented by truncation automata get enforced at runtime [28, 1, 2]; Dam, Jacobs, Lundblad, and Piessens found that separating monitorable multithreaded application code from unmonitorable Java API code prevents inlined truncation automata from enforcing some safety properties they could otherwise enforce [8]; Le Guernic et al. and Hamlen et al. considered some enforcement capabilities of security automata with access to source code [20, 19, 15]; Ligatti, Bauer, and Walker defined enforcement in the presence of execution-transforming monitors called edit automata and found that edit automata enforce a set of policies called the renewal properties [23]; Talhi, Tawbi, and Debbabi considered the enforcement powers of memory-constrained truncation and edit automata [26]; and Falcone, Fernandez, and Mounier compared the set of properties enforceable with edit-automata-like monitors with the safety-progress hierarchy of properties (as defined by Chang, Manna, and Pnueli [6]) [13]. In addition, security automata have formed the basis of several policy-specification languages [10, 9, 5, 2].

Some of the security-automata-related research mentioned above has considered results of actions [1, 2, 8], but in all these cases the models have been constructed to handle the specific case of monitoring Java API methods invoked by Java applications. Also, these previous models treat results as kinds of actions. Treating the return of a result as an action poses no problems in the earlier models because those models only consider monitors as truncation automata, which always accept valid actions (and results) and halt the target immediately upon encountering an invalid action (or result). In contrast, this paper gives monitors the practical ability to edit (i.e., transform) streams of actions and results, which forces us to define a new operational semantics for security automata that carefully distinguishes between how the automata can edit actions and results (because the monitors can freely output any number of actions for every action input but may only output at most one result for every action input and cannot input a new action until a result for the current input action has been output).

7 Conclusions

This paper has presented a framework for reasoning about runtime enforcement. It is the first framework we know of that models monitors transforming actions and results to enforce their validity, while obeying the realistic constraint that a result must be returned for the current (synchronous) action before a new action can be generated. Incorporating results into the model has forced us to create novel notation and definitions (of executions, policies, properties, monitor configurations, and how runtime mechanisms can transform input executions into output executions to enforce policies). Analyzing these new definitions sheds light on the sorts of policies runtime monitors can enforce; we have proved that monitors modeled by MRAs enforce safety properties in highly uncertain environments and no less than action-life properties in more certain environments.

The most surprising results of this study have been:

- Mechanisms that monitor both actions and results may build input and output executions in nontrivial ways.
- Executions input to and output from MRAs may contain holes where actions or events would normally occur.
- Policies and properties in systems with MRAs need to specify whether hole-containing executions are valid.
- MRAs can faithfully model practical monitors while maintaining a compact (four-rule) operational semantics.
- $\text{MRAs}_{\text{exit}, \leftarrow}$ can enforce a rich set of properties, including some—such as a limited kind of action-termination property—that remarkably lie outside the ω -safety properties.

These findings, and theories of runtime enforcement in general, are important because they shape how we think about the roles and meanings of mechanisms, policies, and enforcement, help us reason about whether specific mechanisms enforce desired policies, influence our decisions about how to specify policies and mechanisms (including designs of policy-specification languages and tools), make us aware of the security implications of providing more certain environments for mechanisms to execute in, and improve our understanding of which policies can and cannot be enforced at runtime.

Acknowledgments This research was supported by NSF 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] I. Aktug and K. Naliuka. ConSpec—a formal language for policy specification. In *Proceedings of the First International Workshop on Run Time Enforcement for Mobile and Distributed Systems*, Sept. 2007.
- [3] B. Alpern and F. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1987.
- [4] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, Oct. 1985.
- [5] L. Bauer, J. Ligatti, and D. Walker. Composing expressive runtime security policies. *ACM Transactions on Software Engineering and Methodology*, 18(3):1–43, 2009.
- [6] E. Chang, Z. Manna, and A. Pnueli. Characterization of temporal property classes. In *Proceedings of Automata, Languages and Programming, volume 623 of LNCS*, pages 474–486. Springer-Verlag, 1992.

- [7] M. R. Clarkson and F. B. Schneider. Hyperproperties. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium*, pages 51–65, 2008.
- [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] Ú. Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Cornell University, Jan. 2004.
- [10] Ú. Erlingsson and F. B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the New Security Paradigms Workshop*, pages 87–95, Caledon Hills, Canada, Sept. 1999.
- [11] Ú. Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2000.
- [12] D. Evans and A. Twyman. Flexible policy-directed code safety. In *IEEE Security and Privacy*, Oakland, CA, May 1999.
- [13] Y. Falcone, J.-C. Fernandez, and L. Mounier. Enforcement monitoring wrt. the safety-progress classification of properties. In *Proceedings of the 24th Annual ACM Symposium on Applied Computing—Software Verification and Testing Track (SAC-SVT)*, 2009.
- [14] P. W. L. Fong. Access control by tracking shallow execution history. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2004.
- [15] 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.
- [16] 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.
- [17] 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.
- [18] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions of Software Engineering*, 3(2):125–143, 1977.
- [19] G. Le Guernic. Automaton-based confidentiality monitoring of concurrent programs. In *Proceedings of the Computer Security Foundations Symposium (CSF)*, pages 218–232, July 2007.
- [20] G. Le Guernic, A. Banerjee, T. Jensen, and D. A. Schmidt. Automata-based confidentiality monitoring. In *Proceedings of the Asian Computing Science Conference (ASIAN)*, Dec. 2006.
- [21] J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. Technical Report TR-681-03, Princeton University, May 2003.
- [22] J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1–2):2–16, Feb. 2005.
- [23] 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.
- [24] 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.
- [25] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and Systems Security*, 3(1):30–50, Feb. 2000.

- [26] C. Talhi, N. Tawbi, and M. Debbabi. Execution monitoring enforcement under memory-limitation constraints. *Information and Computation*, 206(2–4):158–184, 2008.
- [27] M. Viswanathan. *Foundations for the Run-time Analysis of Software Systems*. PhD thesis, University of Pennsylvania, 2000.
- [28] D. Walker. A type system for expressive security policies. In *Twenty-Seventh ACM Symposium on Principles of Programming Languages*, pages 254–267, Boston, Jan. 2000.

A Proofs

Lemma 2. For all $M = (E, Q, q_0, \delta)$, $q, q' \in Q$, $\alpha_i, \alpha_o, \alpha'_i, \alpha'_o \in \dot{A}$, $\rho_i, \rho_o, \rho'_i, \rho'_o \in \dot{R}$, and $\varepsilon_T, \varepsilon_S, x_T, x_S \in E^*$:

$$\left(\begin{array}{c} \frac{\alpha_i [q]_{\rho_i}^{\alpha_o} \xrightarrow[\varepsilon_S]{\varepsilon_T} \alpha'_i [q']_{\rho'_i}^{\alpha'_o} \wedge \frac{\alpha_i [q]_{\rho_i}^{\alpha_o} \xrightarrow[x_S]{x_T} \alpha'_i [q']_{\rho'_i}^{\alpha'_o}}{\text{actions}(\varepsilon_T) = \text{actions}(x_T) \wedge \text{results}(\varepsilon_S) = \text{results}(x_S)} \end{array} \right) \implies (\varepsilon_T = x_T \wedge \varepsilon_S = x_S)$$

Proof. By induction on the derivation of $\frac{\alpha_i [q]_{\rho_i}^{\alpha_o} \xrightarrow[\varepsilon_S]{\varepsilon_T} \alpha'_i [q']_{\rho'_i}^{\alpha'_o}}$.

Case Reflexive: $\frac{\alpha_i [q]_{\rho_i}^{\alpha_o} \longrightarrow \alpha_i [q]_{\rho_i}^{\alpha_o}}$

By assumption, $\varepsilon_T = \cdot$, $\varepsilon_S = \cdot$, $\frac{\alpha_i [q]_{\rho_i}^{\alpha_o} \xrightarrow[x_S]{x_T} \alpha_i [q]_{\rho_i}^{\alpha_o}}$, $\text{actions}(\varepsilon_T) = \text{actions}(x_T)$, and $\text{results}(\varepsilon_S) = \text{results}(x_S)$. Therefore $x_T = \cdot$ and $x_S = \cdot$, and the lemma holds.

$$\frac{\frac{\alpha_i [q]_{\rho_i}^{\alpha_o} \xrightarrow[\varepsilon'_S]{\varepsilon'_T} \alpha'_i [q']_{\rho'_i}^{\alpha'_o} \quad \frac{\alpha'_i [q']_{\rho'_i}^{\alpha'_o} \xrightarrow[\varepsilon''_S]{\varepsilon''_T} \alpha''_i [q'']_{\rho''_i}^{\alpha''_o}}{\text{actions}(\varepsilon'_T) = \text{actions}(x'_T) \wedge \text{results}(\varepsilon'_S) = \text{results}(x'_S)}}{\alpha_i [q]_{\rho_i}^{\alpha_o} \xrightarrow[\varepsilon'_S; \varepsilon''_S]{\varepsilon'_T; \varepsilon''_T} \alpha''_i [q'']_{\rho''_i}^{\alpha''_o}}$$

Case Transitive:

1. $\frac{\alpha'_i [q']_{\rho'_i}^{\alpha'_o} \xrightarrow[\varepsilon''_S]{\varepsilon''_T} \alpha''_i [q'']_{\rho''_i}^{\alpha''_o}}$ By assumption
2. $\varepsilon_T = \varepsilon'_T; \varepsilon''_T \wedge \varepsilon_S = \varepsilon'_S; \varepsilon''_S$ By assumption
3. $\frac{\alpha_i [q]_{\rho_i}^{\alpha_o} \xrightarrow[x_S]{x_T} \alpha''_i [q'']_{\rho''_i}^{\alpha''_o}}$ By assumption
4. $\text{actions}(\varepsilon_T) = \text{actions}(x_T) \wedge \text{results}(\varepsilon_S) = \text{results}(x_S)$ By assumption

By definition of multi-step and 3 we have the following two cases (5 and 6 below).

5. Case Reflexive: $\frac{\alpha_i [q]_{\rho_i}^{\alpha_o} \longrightarrow \alpha_i [q]_{\rho_i}^{\alpha_o}}$

- (a) In this case $x_T = x_S = \cdot$. Because $\text{actions}(\varepsilon_T) = \text{actions}(x_T)$ we have $\varepsilon_T = x_T = \cdot$.

- (b) $results(\varepsilon_S) = results(x_S) = \cdot$. By 4 ε_S could be an action. But this cannot happen because the rule *Output-Act* could not have been applied because α_o does not change in the above transition (i.e., reflexive multi-step). So nothing has been input or output; therefore $\varepsilon_S = \cdot$ and $\varepsilon_S = x_S$. Hence the lemma holds in this case.

$$\frac{\begin{array}{c} \alpha_i [q]_{\rho_i}^{\alpha_o} \xrightarrow[x'_S]{x'_T} \alpha_i''' [q''']_{\rho_i'''}^{\alpha_o'''} \quad \alpha_i''' [q''']_{\rho_i'''}^{\alpha_o'''} \xrightarrow[x''_S]{x''_T} \alpha_i'' [q'']_{\rho_i''}^{\alpha_o''} \end{array}}{\alpha_i [q]_{\rho_i}^{\alpha_o} \xrightarrow[x'_S; x''_S]{x'_T; x''_T} \alpha_i'' [q'']_{\rho_i''}^{\alpha_o''}}$$

6. Case Transitive:

- (a) $\alpha_i [q]_{\rho_i}^{\alpha_o} \xrightarrow[x'_S; x''_S]{x'_T; x''_T} \alpha_i'' [q'']_{\rho_i''}^{\alpha_o''}$ By assumption
- (b) $\varepsilon'_T = x'_T, \varepsilon'_S = x'_S, \alpha_i''' = \alpha'_i, \alpha_o''' = \alpha'_o, \rho_i''' = \rho'_i, \rho_o''' = \rho'_o$, and $q''' = q'$ Because δ of an MRA is a deterministic function, its output is the same for the same input.
- (c) $\alpha'_i [q']_{\rho'_i}^{\alpha'_o} \xrightarrow[x''_S]{x''_T} \alpha_i'' [q'']_{\rho_i''}^{\alpha_o''}$ By 6(b) and the second premise of the transitive rule above
- (d) $x_T = x'_T; x''_T \wedge x_S = x'_S; x''_S$ By 6(a) and 3
- (e) $x_T = \varepsilon'_T; x''_T \wedge x_S = \varepsilon'_S; x''_S$ By 6(b) and 6(d)
- (f) $actions(\varepsilon'_T) = actions(x'_T)$ By 2, 4, and 6(e)
- (g) $results(\varepsilon'_S) = results(x'_S)$ By 2, 4, and 6(e)
- (h) $x''_T = \varepsilon''_T$ and $x''_S = \varepsilon''_S$ By 1, 6(c), 6(f), 6(g), and the induction hypothesis
- (i) $x_T = \varepsilon_T$ and $x_S = \varepsilon_S$ By 2, 6(e), and 6(h)
- Hence the lemma holds.

■

Lemma 3. For all MRAs $M = (E, Q, q_0, \delta)$ and $\varepsilon_T, \varepsilon_S, x_T, x_S \in E^\infty$: if $\varepsilon_T \xleftrightarrow[M]{\leftarrow} \varepsilon_S$, $x_T \xleftrightarrow[M]{\leftarrow} x_S$, $actions(\varepsilon_T) = actions(x_T)$, and $results(\varepsilon_S) = results(x_S)$ then $\varepsilon_T = x_T$ and $\varepsilon_S = x_S$.

Proof. By Definition 1 we know that starting in its initial state, M builds ε_T and ε_S by building target and system executions containing every prefix of ε_T and ε_S , and conversely M only builds target and system executions that are prefixes of ε_T and ε_S when receiving input events from ε_T and ε_S . Similarly for x_T and x_S . We are given that $actions(\varepsilon_T) = actions(x_T)$ and $results(\varepsilon_S) = results(x_S)$ which implies that the inputs to the monitor are same. From Lemma 2 we know that all prefixes of the system and target executions are the same for a given sequence of inputs. Therefore ε_T and x_T , and ε_S and x_S , share all the same prefixes, so $\varepsilon_T = x_T$ and $\varepsilon_S = x_S$. ■

Lemma 4. For all event sets E and $\varepsilon_T, \varepsilon_S \in E^*$ and $\varepsilon_i, \varepsilon_o \in E_\square^*$, if $input_*(\varepsilon_T, \varepsilon_S) = \varepsilon_i$ and $output_*(\varepsilon_T, \varepsilon_S) = \varepsilon_o$ then $actions(\varepsilon_T) = actions(\varepsilon_i)$, $results(\varepsilon_S)$ is some permutation of $results(\varepsilon_i)$, $actions(\varepsilon_S) = actions(\varepsilon_o)$, and $results(\varepsilon_T)$ is some permutation of $results(\varepsilon_o)$.

Proof. By induction on the derivations of $input_*(\varepsilon_T, \varepsilon_S) = \varepsilon_i$ and $output_*(\varepsilon_T, \varepsilon_S) = \varepsilon_o$.

Case: $\overline{input_*(\alpha_T, \alpha_S)} = \alpha_T$

Here $\varepsilon_T = \alpha_T$ and $\varepsilon_i = \alpha_T$. Therefore $actions(\varepsilon_T) = actions(\varepsilon_i) = \alpha_T$ and $results(\varepsilon_S) = results(\varepsilon_i) = \cdot$.

$$\text{Case: } \frac{a \notin \varepsilon'_T \quad \text{input}_*(\varepsilon'_T, \varepsilon'_S) = \varepsilon'_i}{\text{input}_*(\varepsilon'_T, a; r; \varepsilon'_S) = \square; r; \varepsilon'_i}$$

1. $\text{input}_*(\varepsilon'_T, \varepsilon'_S) = \varepsilon'_i$ By assumption
2. $\text{actions}(\varepsilon'_T) = \text{actions}(\varepsilon'_i)$ By 1 and induction hypothesis
3. $\text{results}(\varepsilon'_S)$ is some permutation of $\text{results}(\varepsilon'_i)$ By 1 and induction hypothesis
4. $\varepsilon_T = \varepsilon'_T$ and $\varepsilon_S = a; r; \varepsilon'_S$ By assumption
5. $\varepsilon_i = \square; r; \varepsilon'_i$ By assumption
6. $\text{actions}(\varepsilon_T) = \text{actions}(\varepsilon_i)$ By 4, 5, and 2
7. $\text{results}(\varepsilon_S)$ is some permutation of $\text{results}(\varepsilon_i)$ By 4, 5, and 3
Result is from 6 and 7.

$$\text{Case: } \frac{\forall a \preceq \varepsilon'_S : a \in \varepsilon'_T \quad a' \notin \varepsilon'_S \quad \text{input}_*(\varepsilon'_T, \varepsilon'_S) = \varepsilon'_i}{\text{input}_*(a'; r; \varepsilon'_T, \varepsilon'_S; \alpha) = a'; \square; \varepsilon'_i}$$

1. $\text{input}_*(\varepsilon'_T, \varepsilon'_S) = \varepsilon'_i$ By assumption
2. $\text{actions}(\varepsilon'_T) = \text{actions}(\varepsilon'_i)$ By 1 and induction hypothesis
3. $\text{results}(\varepsilon'_S)$ is some permutation of $\text{results}(\varepsilon'_i)$ By 1 and induction hypothesis
4. $\varepsilon_T = a'; r; \varepsilon'_T$ and $\varepsilon_S = \varepsilon'_S; \alpha$ By assumption
5. $\varepsilon_i = a'; \square; \varepsilon'_i$ By assumption
6. $\text{actions}(\varepsilon_T) = \text{actions}(\varepsilon_i)$ By 4, 5, and 2
7. $\text{results}(\varepsilon_S)$ is some permutation of $\text{results}(\varepsilon_i)$ By 4, 5, and 3
Result is from 6 and 7.

$$\text{Case: } \frac{\forall a \preceq \varepsilon'_S : a \in \varepsilon'_T \quad a' \notin \varepsilon'_S \quad \text{input}_*(\varepsilon'_T, \varepsilon'_S; \varepsilon''_S) = \varepsilon'_i}{\text{input}_*(a'; \rho; \varepsilon'_T, \varepsilon'_S; a' r'; \varepsilon''_S) = a'; r'; \varepsilon'_i}$$

1. $\text{input}_*(\varepsilon'_T, \varepsilon'_S; \varepsilon''_S) = \varepsilon'_i$ By assumption
2. $\text{actions}(\varepsilon'_T) = \text{actions}(\varepsilon'_i)$ By 1 and induction hypothesis
3. $\text{results}(\varepsilon'_S; \varepsilon''_S)$ is some permutation of $\text{results}(\varepsilon'_i)$ By 1 and induction hypothesis
4. $\varepsilon_T = a'; \rho; \varepsilon'_T$ and $\varepsilon_S = \varepsilon'_S; a'; r'; \varepsilon''_S$ By assumption
5. $\varepsilon_i = a'; r'; \varepsilon'_i$ By assumption
6. $\text{actions}(\varepsilon_T) = \text{actions}(\varepsilon_i)$ By 4, 5, and 2
7. $\text{results}(\varepsilon_S)$ is some permutation of $\text{results}(\varepsilon_i)$ By 4, 5, and 3
Result is from 6 and 7.

Hence the lemma holds in all the cases. The proof for output_* is similar. ■

Lemma 5. For all event sets E , target and system executions $\varepsilon_T, \varepsilon_S \in E^\infty$, actions $a \in A$, and results $r \in R$:

- $input_*(\varepsilon_T, \varepsilon_S) \preceq_{\square} input_*(\varepsilon_T; a, \varepsilon_S)$
- $output_*(\varepsilon_T, \varepsilon_S) = output_*(\varepsilon_T; a, \varepsilon_S)$
- $input_*(\varepsilon_T, \varepsilon_S) = input_*(\varepsilon_T; r, \varepsilon_S)$
- $output_*(\varepsilon_T, \varepsilon_S) \preceq_{\square} output_*(\varepsilon_T; r, \varepsilon_S)$
- $input_*(\varepsilon_T, \varepsilon_S) = input_*(\varepsilon_T, \varepsilon_S; a)$
- $output_*(\varepsilon_T, \varepsilon_S) \preceq_{\square} output_*(\varepsilon_T, \varepsilon_S; a)$
- $input_*(\varepsilon_T, \varepsilon_S) \preceq_{\square} input_*(\varepsilon_T, \varepsilon_S; r)$
- $output_*(\varepsilon_T, \varepsilon_S) = output_*(\varepsilon_T, \varepsilon_S; r)$

Proof. Let $input_*(\varepsilon_T, \varepsilon_S) = \varepsilon_i$ and $output_*(\varepsilon_T, \varepsilon_S) = \varepsilon_o$. Then there are two cases; in the first we append an event to ε_T , and in the second we append an event to ε_S .

1. If an event e is appended to ε_T then there are two subcases: $e = a$ and $e = r$. We show that the lemma holds in both the sub-cases.
 - (a) In this case we show that the lemma holds when $e = a$. First, $output_*(\varepsilon_T; a, \varepsilon_S) = output_*(\varepsilon_T, \varepsilon_S)$ by induction on the derivation of $output_*(\varepsilon_T; a, \varepsilon_S)$. Second, by Lemma 4, $actions(\varepsilon_T) = actions(input_*(\varepsilon_T, \varepsilon_S))$ and $actions(\varepsilon_T; a) = actions(input_*(\varepsilon_T; a, \varepsilon_S)) = actions(\varepsilon_T); a$. Therefore, appending a to ε_T causes a to be appended to the sequence of actions in ε_i , which can happen when (1) ε_i is natural and $input_*(\varepsilon_T; a, \varepsilon_S) = \varepsilon_i; a$ or (2) $\varepsilon_i = \varepsilon; \square; r_1; \square; r_2; \square; r_3; \dots$ and ε is natural and $input_*(\varepsilon_T; a, \varepsilon_S) = \varepsilon; a; r_1; \square; r_2; \square; r_3; \dots$. Therefore $input_*(\varepsilon_T, \varepsilon_S) \preceq_{\square} input_*(\varepsilon_T; a, \varepsilon_S)$ and $output_*(\varepsilon_T, \varepsilon_S) = output_*(\varepsilon_T; a, \varepsilon_S)$.
 - (b) In this case we show that the lemma holds when $e = r$. First, $input_*(\varepsilon_T; r, \varepsilon_S) = input_*(\varepsilon_T, \varepsilon_S)$ by induction on the derivation of $input_*(\varepsilon_T; r, \varepsilon_S)$. Second, by Lemma 4, $results(\varepsilon_T)$ is some permutation of $results(output_*(\varepsilon_T, \varepsilon_S))$ and $results(\varepsilon_T; r)$ is some permutation of $results(output_*(\varepsilon_T; r, \varepsilon_S))$. Therefore, appending r to ε_T causes r to be combined with the sequence of results in ε_o , which can happen when (1) ε_o is natural and $output_*(\varepsilon_T; r, \varepsilon_S) = \varepsilon_o; \square; r$ or (2) $\varepsilon_o = \varepsilon; a; \square; \varepsilon'$ and $output_*(\varepsilon_T; r, \varepsilon_S) = \varepsilon; a; r; \varepsilon'$. Therefore $input_*(\varepsilon_T, \varepsilon_S) = input_*(\varepsilon_T; r, \varepsilon_S)$ and $output_*(\varepsilon_T, \varepsilon_S) \preceq_{\square} output_*(\varepsilon_T; r, \varepsilon_S)$.
2. If an event e is added to ε_S then there are two subcases $e = a$ and $e = r$. We show that the lemma holds in both the sub-cases.
 - (a) In this case we show that the lemma holds when $e = a$. First, $input_*(\varepsilon_T, \varepsilon_S; a) = input_*(\varepsilon_T, \varepsilon_S)$ by induction on the derivation of $input_*(\varepsilon_T, \varepsilon_S; a)$. Second, by Lemma 4, $actions(\varepsilon_S) = actions(output_*(\varepsilon_T, \varepsilon_S))$ and $actions(\varepsilon_S; a) = actions(output_*(\varepsilon_T, \varepsilon_S; a)) = actions(\varepsilon_S); a$. Therefore, appending a to ε_S causes a to be appended to the sequence of actions in ε_o , which can happen when (1) ε_o is natural and $output_*(\varepsilon_T, \varepsilon_S; a) = \varepsilon_o; a$ or (2) $\varepsilon_o = \varepsilon; \square; r_1; \square; r_2; \square; r_3; \dots$ and ε is natural and $output_*(\varepsilon_T, \varepsilon_S; a) = \varepsilon; a; r_1; \square; r_2; \square; r_3; \dots$. Therefore $input_*(\varepsilon_T, \varepsilon_S) = input_*(\varepsilon_T, \varepsilon_S; a)$ and $output_*(\varepsilon_T, \varepsilon_S) \preceq_{\square} output_*(\varepsilon_T, \varepsilon_S; a)$.
 - (b) In this case we show that the lemma holds when $e = r$. First, $output_*(\varepsilon_T, \varepsilon_S; r) = output_*(\varepsilon_T, \varepsilon_S; r)$ by induction on the derivation of $output_*(\varepsilon_T, \varepsilon_S; r)$. Second, by Lemma 4, $results(\varepsilon_S)$ is some permutation of $results(input_*(\varepsilon_T, \varepsilon_S))$ and $results(\varepsilon_S; r)$ is some permutation of $results(input_*(\varepsilon_T, \varepsilon_S; r))$. Therefore, appending r to ε_S causes r to be combined with the sequence of results

in ε_i , which can happen when (1) ε_i is natural and $input_*(\varepsilon_T, \varepsilon_S; r) = \varepsilon_i; \square; r$ or (2) $\varepsilon_i = \varepsilon; a; \square; \varepsilon'$ and $input_*(\varepsilon_T, \varepsilon_S; r) = \varepsilon; a; r; \varepsilon'$. Therefore $input_*(\varepsilon_T, \varepsilon_S) \preceq_{\square} input_*(\varepsilon_T, \varepsilon_S; r)$ and $output_*(\varepsilon_T, \varepsilon_S) = output_*(\varepsilon_T, \varepsilon_S; r)$.

■

Lemma 6. *For all event sets E and $\varepsilon \in E^*$, $input_*(\varepsilon, \varepsilon) = output_*(\varepsilon, \varepsilon) = \varepsilon$.*

Proof. First we prove $input_*(\varepsilon, \varepsilon) = \varepsilon$. This is proved by induction on the structure of ε . If $\varepsilon = \cdot$ then from the definition of the rule *Input-Basis* we know that $input_*(\varepsilon, \varepsilon) = \cdot$ and hence the lemma holds. If $\varepsilon = a$ then from the definition of the rule *Input-Basis* we know that $input_*(\varepsilon, \varepsilon) = a$ and hence the lemma holds. The induction hypothesis says that $input_*(\varepsilon, \varepsilon) = \varepsilon$. Next we need to show that $input_*(a; r; \varepsilon; a; r; \varepsilon) = a; r; \varepsilon$. From rule *Input-Act-and-Res* we know $input_*(a; r; \varepsilon; a; r; \varepsilon) = a; r; input_*(\varepsilon, \varepsilon)$. From the induction hypothesis we know that $input_*(\varepsilon, \varepsilon) = \varepsilon$. Therefore $input_*(a; r; \varepsilon; a; r; \varepsilon) = a; r; \varepsilon$, as required. The proof that $output_*(\varepsilon, \varepsilon) = \varepsilon$ is similar. ■

Lemma 7. *For all event sets E , $\varepsilon \in E^*$, $a \in A$, and $r \in R$: $input_*(\varepsilon; a, \varepsilon) = \varepsilon; a$ and $output_*(\varepsilon; a, \varepsilon) = \varepsilon$ and $input_*(\varepsilon, \varepsilon; r) = \varepsilon; r$ and $output_*(\varepsilon, \varepsilon; r) = \varepsilon$.*

Proof. We prove $input_*(\varepsilon; a, \varepsilon) = \varepsilon; a$. The proofs that $output_*(\varepsilon; a, \varepsilon) = \varepsilon$ and $input_*(\varepsilon, \varepsilon; r) = \varepsilon; r$ and $output_*(\varepsilon, \varepsilon; r) = \varepsilon$ all proceed similarly. We prove $input_*(\varepsilon; a, \varepsilon) = \varepsilon; a$ by induction on the structure of ε . If $\varepsilon = \cdot$ then from the definition of the rule *Input-Basis* we know that $input_*(\varepsilon; a, \varepsilon) = a$ and hence the lemma holds. The case $\varepsilon = a'$ cannot happen because $a'; a$ is not a valid execution. The induction hypothesis says that $input_*(\varepsilon; a, \varepsilon) = \varepsilon; a$. Next we need to show that $input_*(a'; r'; \varepsilon; a, a'; r'; \varepsilon) = a'; r'; \varepsilon; a$. From rule *Input-Act-and-Res* we know $input_*(a'; r'; \varepsilon; a, a'; r'; \varepsilon) = a'; r'; input_*(\varepsilon; a, \varepsilon)$. From the induction hypothesis we know that $input_*(\varepsilon; a, \varepsilon) = \varepsilon; a$. Therefore $input_*(a'; r'; \varepsilon; a, a'; r'; \varepsilon) = a'; r'; \varepsilon; a$, as required. ■

Lemma 8. *The $input_*$ and $output_*$ relations are total functions.*

Proof. The proof that $input_*$ and $output_*$ are functions is by induction on the derivation of $input_*(\varepsilon_T, \varepsilon_S) = \varepsilon_i$ and $output_*(\varepsilon_T, \varepsilon_S) = \varepsilon_o$ (for some ε_i and ε_o). To prove that $input_*$ is a function we need to show that $\forall \varepsilon_i, x_i : (input_*(\varepsilon_T, \varepsilon_S) = \varepsilon_i \wedge input_*(\varepsilon_T, \varepsilon_S) = x_i) \implies (\varepsilon_i = x_i)$.

Case: $\overline{input_*(\alpha_T, \alpha_S) = \alpha_T}$

Here $\varepsilon_T = \alpha_T$ and $\varepsilon_i = \alpha_T$. By inspecting the rules defining the $input_*$ judgement we know that $input_*(\alpha_T, \alpha_S) = \alpha_T$ always. Hence the lemma holds in this case.

$\overline{a \notin \varepsilon_T \quad input_*(\varepsilon_T, \varepsilon_S) = \varepsilon_i}$

Case: $input_*(\varepsilon_T, a; r; \varepsilon_S) = \square; r; \varepsilon_i$

Depending on the value of ε_T we have two subcases:

- (a) If $\varepsilon_T = \cdot$ then by assumption we know that $input_*(\cdot, \varepsilon_S) = \varepsilon_i$. By inspection of rules defining $input_*$ we observe that rule (*Input-Basis*) cannot apply (for determining $input_*(\cdot, a; r; \varepsilon_S)$) because the arguments of $input_*$ are not of the form α_T and α_S . Rules (*Input-No-Res*) and (*Input-Act-and-Res*) also cannot apply because the first argument of $input_*$ cannot be \cdot for these rules to be applied. Hence rule *Input-No-Act* is the only rule that applies in this case and $input_*(\cdot, a; r; \varepsilon_S) = \square; r; input_*(\cdot, \varepsilon_S) = \square; r; \varepsilon_i$.
- (b) If $\varepsilon_T \neq \cdot$ then by assumption we know that $input_*(\varepsilon_T, \varepsilon_S) = \varepsilon_i$. By inspection of the rules defining $input_*$ we observe that none besides *Input-No-Act* apply (for determining $input_*(\varepsilon_T, a; r; \varepsilon_S)$) because their premises contradict the assumptions of rule *Input-No-Act*. Hence rule *Input-No-Act* is the only

rule that applies in this case and $input_*(\varepsilon_T, a; r; \varepsilon_S) = \square; r; input_*(\varepsilon_T, \varepsilon_S) = \square; r; \varepsilon_i$.

$$\frac{\forall a \preceq \varepsilon_S : a \in \varepsilon_T \quad a' \notin \varepsilon_S \quad input_*(\varepsilon_T, \varepsilon_S) = \varepsilon_i}{Case: \quad input_*(a'; r; \varepsilon_T, \varepsilon_S; \alpha) = a'; \square; \varepsilon_i}$$

Depending on the value of ε_S we have two subcases:

- (a) If $\varepsilon_S = \cdot$ then by assumption we know that $input_*(\varepsilon_T, \cdot) = \varepsilon_i$. By inspection of rules defining $input_*$ we observe that rule (*Input-Basis*) cannot apply (for determining $input_*(a'; r; \varepsilon_T, \alpha)$) because the arguments of $input_*$ are not of the form α_T and α_S . Rules (*Input-No-Act*) and (*Input-Act-and-Res*) also cannot apply because the second argument of $input_*$ cannot be \cdot or just an action for these rules to be applied. Hence rule *Input-No-Res* is the only rule that applies in this case and $input_*(a'; r; \varepsilon_T, \alpha) = a'; \square; input_*(\varepsilon_T, \cdot) = a'; \square; \varepsilon_i$.
- (b) If $\varepsilon_S \neq \cdot$ then by assumption we know that $input_*(\varepsilon_T, \varepsilon_S) = \varepsilon_i$. By inspection of rules defining $input_*$ we observe that rule (*Input-Basis*) cannot apply (for determining $input_*(a'; r; \varepsilon_T, \varepsilon_S; \alpha)$) because the arguments of $input_*$ are not of the form α_T and α_S . Rule (*Input-No-Act*) cannot apply because of the assumption that $\forall a \preceq \varepsilon_S : a \in \varepsilon_T$. Rule (*Input-Act-and-Res*) cannot apply because we know by assumption of *Input-No-Res* that $a' \notin \varepsilon_S$, but the rule *Input-Act-and-Res* requires a' to be present in the second argument of $input_*$. Hence rule *Input-No-Res* is the only rule that applies in this case and $input_*(a'; r; \varepsilon_T, \varepsilon_S; \alpha) = a'; \square; input_*(\varepsilon_T, \varepsilon_S) = a'; \square; \varepsilon_i$.

$$\frac{\forall a \preceq \varepsilon_S : a \in \varepsilon_T \quad a' \notin \varepsilon_S \quad input_*(\varepsilon_T, \varepsilon_S; \varepsilon'_S) = \varepsilon_i}{Case: \quad input_*(a'; \rho; \varepsilon_T, \varepsilon_S; a'; r'; \varepsilon'_S) = a'; r'; \varepsilon_i}$$

No matter what the values of ε_T , ε_S , ε'_S , and ρ are, by inspection of rules defining $input_*$ we observe that rule (*Input-Basis*) cannot apply (for determining $input_*(a'; \rho; \varepsilon_T, \varepsilon_S; a'; r'; \varepsilon'_S)$) because the arguments of $input_*$ are not of the form α_T and α_S . Rule (*Input-No-Act*) cannot apply because of the assumption that $\forall a \preceq \varepsilon_S : a \in \varepsilon_T$, but for rule *Input-No-Act* to apply we need $a \notin \varepsilon_T$ if $\varepsilon_S \neq \cdot$. If $\varepsilon_S = \cdot$ then $input_*(a'; \rho; \varepsilon_T, a'; r'; \varepsilon'_S)$ also violates the condition specified in rule *Input-No-Act* (that $a' \notin a'; \rho; \varepsilon_T$). Rule (*Input-No-Res*) cannot apply because it requires a' not to be present in the second argument of $input_*$, but in this case a' is present in the second argument of $input_*$ at all times.

Hence, in all cases, the $input_*$ relation is a function.

We prove that $input_*$ is total by strong induction on the sum of the lengths of ε_T and ε_S . If $\varepsilon_T = \cdot$ and $\varepsilon_S = \cdot$ then from the definition of the rule *Input-Basis* we know that $input_*(\varepsilon_T, \varepsilon_S) = \cdot$. If $\varepsilon_T = a$ and $\varepsilon_S = \cdot$ then from the definition of the rule *Input-Basis* we know that $input_*(\varepsilon_T, \varepsilon_S) = a$. If $\varepsilon_T = \cdot$ and $\varepsilon_S = a$ then from the definition of the rule *Input-Basis* we know that $input_*(\varepsilon_T, \varepsilon_S) = \cdot$. And if $\varepsilon_T = a_T$ and $\varepsilon_S = a_S$ then from the definition of the rule *Input-Basis* we know that $input_*(\varepsilon_T, \varepsilon_S) = a_T$. The induction hypothesis says that the $input_*$ function is total for all total lengths of inputs from zero to $|\varepsilon_T| + |\varepsilon_S| - 1$, so because all rules recursively build input executions with target and system executions whose lengths sum to a smaller value than the current sum of the lengths of ε_T and ε_S , if we can show that all possibilities for ε_T and ε_S satisfy the constraints of some rule then we know that $input_*$ is total. Depending on the values of ε_T and ε_S we have the following cases:

Already, we know that rule *Input-Basis* applies when $\varepsilon_T = \alpha_T$ and $\varepsilon_S = \alpha_S$.

If $\varepsilon_S = \cdot$ and $\varepsilon_T \neq \alpha_T$ then rule *Input-No-Res* applies.

Else if $\varepsilon_S = a$ and $\varepsilon_T \neq \alpha_T$ then rule *Input-No-Res* applies.

(at this point we have handled the cases where $\varepsilon_S = \alpha_S$)

Else if $\varepsilon_S = a; r; \varepsilon'_S$ and $a \notin \varepsilon_T$ then rule *Input-No-Act* applies.

Else if $\varepsilon_S = a; r; \varepsilon'_S$ and $\varepsilon_T = a$ then rule *Input-Act-and-Res* applies.

Else if $\varepsilon_S = a; r; \varepsilon'_S$ and $\varepsilon_T = a; r; \varepsilon'_T$ then rule *Input-Act-and-Res* applies.

(at this point we have handled all the cases of ε_T and ε_S , except the ones in which $\varepsilon_S = a; r; \varepsilon'_S, \varepsilon_T = a'; r'; \varepsilon'_T, a \in \varepsilon'_T$ and $a \neq a'$)

Else if $\varepsilon_S = a; r; \varepsilon'_S$, and $\varepsilon_T = a'; r'; \varepsilon'_T, a \in \varepsilon'_T, a' \notin \varepsilon'_S$ and $a \neq a'$ then rule *Input-No-Res* applies.

Else if $\varepsilon_S = a; r; \varepsilon'_S; a'; r''; \varepsilon''_S$ and $\varepsilon_T = a'; r'; \varepsilon'_T, a \in \varepsilon'_T, a' \notin \varepsilon'_S$ and $a \neq a'$ then rule *Input-Act-and-Res* applies.

Else if $\varepsilon_S = a; r; \varepsilon'_S; a'$ and $\varepsilon_T = a'; r'; \varepsilon'_T$ and $a \in \varepsilon'_T$ and $a' \notin \varepsilon'_S$ and $a \neq a'$ then rule *Input-No-Res* applies.

Therefore, no matter what ε_T and ε_S values are given, $input_*(\varepsilon_T, \varepsilon_S)$ is defined, and $input_*$ is a total function. The proof that $output_*$ is a total function is similar. ■

Lemma 11. *The $input_\infty$ and $output_\infty$ relations are total functions.*

Proof. We prove that $input_\infty$ is a function by contradiction. For the sake of obtaining a contradiction assume that $input_\infty(\varepsilon_T, \varepsilon_S) = \varepsilon_i$ and $input_\infty(\varepsilon_T, \varepsilon_S) = \varepsilon'_i$ and $\varepsilon'_i \neq \varepsilon_i$ (for some $\varepsilon_T, \varepsilon_S \in E^\infty$). By Definition 9 we know that every prefix of ε_i gets input from the target execution ε_T and system execution ε_S and only prefixes of ε_i get input from ε_T and ε_S . Similarly, by our assumption every prefix of ε'_i gets input from ε_T and ε_S and only prefixes of ε'_i get input from ε_T and ε_S . Also for the assumption that $\varepsilon_i \neq \varepsilon'_i$ to hold we need that there exist(s) certain prefix(es) of ε_T and ε_S such that $input_*$ returns two inputs for them. But we know that $input_*$ is a total function. This implies that $input_*$ returns the same input for a given set of system and target executions. This contradicts our assumption that $input_\infty$ is not a function. Hence, $input_\infty$ is a function.

Next we prove that $input_\infty$ is total. Given an $\varepsilon_T, \varepsilon_S \in E^\infty$, we define ε_i and show that $input_\infty(\varepsilon_T, \varepsilon_S) = \varepsilon_i$ per the constraints laid out in Definition 9. We define e_j , the j^{th} event in ε_i as follows (under the assumption that all finite executions are made into equivalent infinite-length executions by appending an infinite-length sequence of \square s):

$$e_j = \begin{cases} \text{the } j^{th} \text{ event of } input_*(\varepsilon'_T, \varepsilon'_S) & \text{if the } j^{th} \text{ event of } input_*(\varepsilon'_T, \varepsilon'_S) \neq \square \text{ for some } \varepsilon'_T \preceq \varepsilon_T, \varepsilon'_S \preceq \varepsilon_S \\ \square & \text{if for all } \varepsilon'_T \preceq \varepsilon_T \text{ and } \varepsilon'_S \preceq \varepsilon_S, \text{ the } j^{th} \text{ event of } input_*(\varepsilon'_T, \varepsilon'_S) = \square \end{cases}$$

Now that we have defined the structure of ε_i we prove that it satisfies Definition 9. For Constraint (1) of Definition 9, consider any $x_i \preceq \varepsilon_i$. From the definition of ε_i 's events above, every non- \square event at position j in x_i is the j^{th} event of $input_*(\varepsilon'_T, \varepsilon'_S)$, for some $\varepsilon'_T \preceq \varepsilon_T$ and $\varepsilon'_S \preceq \varepsilon_S$, while every \square at position j in x_i occurs only when the j^{th} event of $input_*(\varepsilon'_T, \varepsilon'_S) = \square$ for all $\varepsilon'_T \preceq \varepsilon_T$ and $\varepsilon'_S \preceq \varepsilon_S$. We also know from Lemma 5 that extending the target and system executions only extends and fills in holes in input executions. Therefore, if we let x_T be the longest ε'_T used to define any event e_j in x_i and x_S be the longest ε'_S used to define any event e_j in x_i , then it must be the case that $x_i \preceq input_*(x_T, x_S)$.

For constraint (2) of Definition 9, consider any $x_T \preceq \varepsilon_T$ and $x_S \preceq \varepsilon_S$. Also consider any $j \in \mathbb{N}$ and let e'_j be the j^{th} event of $input_*(x_T, x_S)$ (again, working under the assumption that all executions are padded with \square s to have infinite length). By construction of ε_i and Lemma 5, $e'_j = \square$ or $e'_j = e_j$ (where e_j is the j^{th} event of ε_i). Therefore, $input_*(x_T, x_S) \preceq \varepsilon_i$. Because Constraints (1) and (2) of Definition 9 hold, $input_\infty$ is total.

The proof that $output_\infty$ is a total function is similar. ■

Theorem 13. *For all MRAs $M = (E, Q, q_0, \delta)$ and $\varepsilon_i, \varepsilon_o, x_o \in E^\infty$, if $\varepsilon_i \Downarrow_M \varepsilon_o$ and $\varepsilon_i \Downarrow_M x_o$ then $\varepsilon_o = x_o$.*

Proof. By Definition 12 we know that if $\varepsilon_i \Downarrow_M \varepsilon_o$ then $\exists \varepsilon_T, \varepsilon_S \in E^\infty : (\varepsilon_T \xleftrightarrow{M} \varepsilon_S \wedge input_\infty(\varepsilon_T, \varepsilon_S) = \varepsilon_i \wedge output_\infty(\varepsilon_T, \varepsilon_S) = \varepsilon_o)$. Also, if $\varepsilon_i \Downarrow_M x_o$ then $\exists \varepsilon'_T, \varepsilon'_S \in E^\infty : (\varepsilon'_T \xleftrightarrow{M} \varepsilon'_S \wedge input_\infty(\varepsilon'_T, \varepsilon'_S) = \varepsilon_i \wedge output_\infty(\varepsilon'_T, \varepsilon'_S) = x_o)$. Therefore $input_\infty(\varepsilon_T, \varepsilon_S) = input_\infty(\varepsilon'_T, \varepsilon'_S)$, so by Lemma 4, $actions(\varepsilon_T) = actions(\varepsilon'_T)$. Next we show that $results(\varepsilon_S) = results(\varepsilon'_S)$. Let us assume for the sake of obtaining a contradiction that $results(\varepsilon_S) \neq results(\varepsilon'_S)$. This implies that $r_1; r_2; r_3; \dots; r_n \preceq results(\varepsilon_S) \wedge r_1; r_2; r_3; \dots; r'_n \preceq results(\varepsilon'_S) \wedge r_n \neq r'_n$ (for some $r_1, r_2, r_3, \dots, r_n, r'_n \in R$). Because $results(\varepsilon_S) = results(\varepsilon'_S)$ until r_n ,

from Lemma 3 we know that M inputs and outputs the same events in the two runs $\varepsilon_T \xleftrightarrow{M} \varepsilon_S$ and $\varepsilon'_T \xleftrightarrow{M} \varepsilon'_S$, up to the point of inputting r_n in ε_S and r'_n in ε'_S . This implies that if the system returned r_n by executing a_n and returned r'_n by executing a'_n then $a_n = a'_n$. Now we have two possibilities: $a_n \in \varepsilon_T$ and $a_n \notin \varepsilon_T$. If $a_n \notin \varepsilon_T$ then from the definition of *Input-No-Act* we know that r_n and r'_n get directly appended to the input. This implies that the input executions in the two runs differ, and irremediably so (by Lemma 5), which contradicts with the lemma's statement. Hence, $results(\varepsilon_S) = results(\varepsilon'_S)$ must hold in this case. In the case where a_n is present in ε_T we know from the rule *Input-Act-and-Res* that the result of a_n in ε_i would have to be both r_n and r'_n , another contradiction. Therefore $results(\varepsilon_S) = results(\varepsilon'_S)$. Now we know that $\varepsilon_T \xleftrightarrow{M} \varepsilon_S \wedge \varepsilon'_T \xleftrightarrow{M} \varepsilon'_S \wedge actions(\varepsilon_T) = actions(\varepsilon'_T) \wedge results(\varepsilon_S) = results(\varepsilon'_S)$, so from Lemma 3 we deduce that $\varepsilon_T = \varepsilon'_T$ and $\varepsilon_S = \varepsilon'_S$. From Lemma 11 we know $output_\infty$ is a total function, which implies that $output_\infty(\varepsilon_T, \varepsilon_S) = output_\infty(\varepsilon'_T, \varepsilon'_S)$. Therefore $\varepsilon_o = x_o$, and the lemma holds. ■

Theorem 15. *A property \hat{P} on a system with event set E can be enforced by some MRA M if and only if:*

1. $\hat{P}(\cdot)$ (Empty-validity)
2. $\forall \varepsilon \in E^\infty : (\neg \hat{P}(\varepsilon) \implies \exists x \preceq \varepsilon : \neg alive(x))$ (Safety)

Proof. For the *if* direction, we construct an MRA M that enforces any such \hat{P} as follows:

- States: $Q = E^*$ (the current output execution)
- Start state: $q_0 = \cdot$ (no events output initially)
- Transition function (for simplicity we write δ in terms of high-level transitions):

Consider processing an input event e in state $q = \varepsilon$.

- (a) If $\hat{P}(\varepsilon; e)$, then output e and continue in state $\varepsilon; e$.
- (b) Otherwise, enter an infinite loop.

M maintains the invariant that its target execution ε_T and system execution ε_S are equal and valid, except that ε_T may end with an invalid action that does not appear at the end of ε_S , and ε_S may end with an invalid result that does not appear at the end of ε_T . M establishes this invariant because $\hat{P}(\cdot)$ and initially $\varepsilon_T = \varepsilon_S = \cdot$, and M maintains the invariant in all of its transitions. Given this invariant and Lemma 7, M also maintains the invariant that its input execution ε_i and output execution ε_o are natural, equal, and valid, except that ε_i may end with an invalid action or an invalid result that does not appear at the end of the always valid ε_o . Let $x \in E^\infty$ be an input execution. If $\hat{P}(x)$ then by the definition of safety, all prefixes of x must be valid, so M must output all prefixes of x and only prefixes of x . Hence, M is transparent. Also, M 's invariant ensures that its finite-length outputs are valid, and if M outputs an infinite-length execution ε_o then ε_o cannot be invalid because an invalid ε_o would (by the definition of safety) have to have a dead prefix (contradicting M 's invariant). Hence, M is sound and transparent and enforces \hat{P} .

For the *only-if* direction, we consider any MRA M that enforces a property \hat{P} and show that the two constraints in the theorem statement must hold. First, $\hat{P}(\cdot)$ because otherwise M could not be sound when applications generate no input actions. Second, \hat{P} must be a safety property. As soon as M encounters an invalid input event, it cannot output it while remaining sound because there may be no additional inputs, and M can only make additional outputs if there are additional inputs. Hence, once M encounters an invalid input event, it must cease to output the input, giving up on outputting any valid extensions of the input, so there must not be any valid extensions of the current input. Moreover, for M to enforce \hat{P} on any invalid, natural, infinite-length input execution x , some (finite) prefix of x must be invalid (otherwise transparency would force M to output every valid prefix of x , i.e. *invalid* x overall, on input x). These constraints imply that all invalid natural executions must have some dead prefix, so \hat{P} must be a safety property. ■

Theorem 16. *A property \hat{P} on a system with event set E can be enforced by some $\text{MRA}_{\text{exit}} M$ if and only if \hat{P} is satisfiable and:*

$$\forall \varepsilon \in E^\infty : \neg \hat{P}(\varepsilon) \implies \left(\left(\exists x \preceq \varepsilon : \neg \text{alive}(x) \right) \vee \left(\exists a \in A : \hat{P}(\varepsilon; a) \right) \right) \quad (\text{ACTION-ESCAPE SAFETY})$$

Proof. For the *if* direction, we construct an $\text{MRA}_{\text{exit}} M$ that enforces any such \hat{P} as follows:

- States: $Q = E^*$ (the current output execution)
- Start state: $q_0 = \cdot$ (no events output initially)
- Transition function (for simplicity we write δ in terms of high-level transitions):

Consider processing an event e in state $q = \varepsilon$.

- (a) If $\hat{P}(\varepsilon; e)$, then output e and continue in state $\varepsilon; e$.
- (b) Else if e is a result r then iterate through A searching for an action a such that $\hat{P}(\varepsilon; r; a)$. Note that this search may not terminate (because A may be countably infinite), in which case M will make no additional inputs or outputs. Also, if the search terminates but no such a is found then enter an infinite loop. If an a is found such that $\hat{P}(\varepsilon; r; a)$ then output r and continue in state $\varepsilon; r$.
- (c) Else iterate through A searching for an action a such that $\hat{P}(\varepsilon; a)$. Note that this search may not terminate, in which case M will make no additional inputs or outputs. Also, if the search terminates but no such a is found then enter an infinite loop. If an a is found such that $\hat{P}(\varepsilon; a)$ then output a and enter state $\varepsilon; a$. Upon receiving any additional inputs, enter an infinite loop.

M maintains an invariant in all high-level states ε it reaches that ε is its output execution, its input execution is either ε or $x; a$ (where $\varepsilon = x; a'$ and $\neg \hat{P}(x; a)$), and either $\hat{P}(\varepsilon)$, or when M is awaiting an action input that must eventually arrive then it is possible that $\neg \hat{P}(\varepsilon)$ but $\exists a \in A : \hat{P}(\varepsilon; a)$. M establishes this invariant because $q_0 = \cdot$, initially no events have been input or output, and either $\hat{P}(\cdot)$ or $\exists a \in A : \hat{P}(a)$ (by a combination of the action-escape-safety constraint and the satisfiability of \hat{P}). M maintains this invariant in all high-level states it reaches. This invariant guarantees that M 's output is always valid except temporarily, when M is about to receive an action a as input but is certain that it can replace a (if it is invalid) with some valid action output, which it will do if necessary using transition (c); hence, M is sound. M is also transparent on any valid input execution x because action-escape safety implies that all prefixes of x must either be valid or become valid when extended by some action, while transitions (a) and (b) ensure that M outputs all such prefixes verbatim.

For the *only-if* direction, we consider any $\text{MRA}_{\text{exit}} M$ that enforces a property \hat{P} . First, \hat{P} must be satisfiable because otherwise M could not be sound (it could never produce a valid output execution for any input execution). Second, \hat{P} must be an action-escape-safety property. As soon as M encounters an invalid input action a , it cannot output a while remaining sound because there may be no additional inputs, and M can only make additional outputs if there are additional inputs. Hence, once M encounters an invalid input action, it must cease to output the input, giving up on transparently outputting any valid extensions of the input, so there must not be any valid extensions of the current input. On the other hand, M may temporarily produce an invalid output execution that does not end with an action, but only if M is assured that it will be able to output some valid action when the next input action arrives (which we know must eventually occur because the current output does not end with an action). Alternatively, when M inputs an invalid execution that does not end with an action, it may diverge at some point in processing its input, giving up on outputting any valid extensions of the non-action-ending input (so there again must not be any valid extensions of the invalid input). Moreover, for M to enforce \hat{P} on any invalid, natural, infinite-length input execution x , some (finite) prefix of x ending with an action must be invalid (otherwise transparency would force M to output every valid prefix of x that ends with an action, i.e. *invalid* x overall, on input

x). Putting all this together, we have requirements on \hat{P} that all invalid, natural, action-ending executions must be dead; a natural non-action-ending execution x may be invalid but only if it is either dead or can be extended with some action a such that $\hat{P}(x; a)$; and for all invalid, natural, infinite-length executions x , there exists some invalid action-ending prefix of x (i.e., there exists a prefix of x that must be dead). These three constraints imply the action-escape-safety constraint in the theorem statement. ■

Theorem 17. *A property \hat{P} on a system with event set E can be enforced by some $\text{MRA}_{\hookrightarrow} M$ if and only if $\hat{P}(\cdot)$ and:*

$$\forall \varepsilon \in E^\infty : \neg \hat{P}(\varepsilon) \implies \left((\exists x \preceq \varepsilon : \neg \text{alive}(x)) \vee (\exists r \in R : \hat{P}(\varepsilon; r)) \right) \quad (\text{RESULT-ESCAPE SAFETY})$$

Proof. For the *if* direction, we construct an $\text{MRA}_{\hookrightarrow} M$ that enforces any such \hat{P} as follows:

- States: $Q = E^*$ (the current output execution)
- Start state: $q_0 = \cdot$ (no events output initially)
- Transition function (for simplicity we write δ in terms of high-level transitions):

Consider processing an event e in state $q = \varepsilon$.

- (a) If $\hat{P}(\varepsilon; e)$, then output e and continue in state $\varepsilon; e$.
- (b) Else if e is an action a then iterate through R searching for a result r such that $\hat{P}(\varepsilon; a; r)$. Note that this search may not terminate (because R may be countably infinite), in which case M will make no additional inputs or outputs. Also, if the search terminates but no such r is found then enter an infinite loop. If an r is found such that $\hat{P}(\varepsilon; a; r)$ then output a and continue in state $\varepsilon; a$.
- (c) Else iterate through R searching for a result r such that $\hat{P}(\varepsilon; r)$. Note that this search may not terminate, in which case M will make no additional inputs or outputs. Also, if the search terminates but no such r is found then enter an infinite loop. If an r is found such that $\hat{P}(\varepsilon; r)$ then output r and enter state $\varepsilon; r$. Upon receiving any additional inputs, enter an infinite loop.

M maintains an invariant in all high-level states ε it reaches that ε is its output execution, its input execution is either ε or $x; r$ (where $\varepsilon = x; r'$ and $\neg \hat{P}(x; r)$), and either $\hat{P}(\varepsilon)$, or when M is awaiting a result input that must eventually arrive then it is possible that $\neg \hat{P}(\varepsilon)$ but $\exists r \in R : \hat{P}(\varepsilon; r)$. M establishes this invariant because $q_0 = \cdot$, initially no events have been input or output, and $\hat{P}(\cdot)$. M maintains this invariant in all high-level states it reaches. This invariant guarantees that M 's output is always valid except temporarily, when M is about to receive a result r as input but is certain that it can replace r (if it is invalid) with some valid result output, which it will do if necessary using transition (c); hence, M is sound. M is also transparent on any valid input execution x because result-escape safety implies that all prefixes of x must either be valid or become valid when extended by some result, while transitions (a) and (b) ensure that M outputs all such prefixes verbatim.

For the *only-if* direction, we consider any $\text{MRA}_{\hookrightarrow} M$ that enforces a property \hat{P} . First, $\hat{P}(\cdot)$ because otherwise M could not be sound when the target generates no input actions. Second, \hat{P} must be a result-escape-safety property. As soon as M encounters an invalid input result r , it cannot output r while remaining sound because there may be no additional inputs, and M can only make additional outputs if there are additional inputs. Hence, once M encounters an invalid input result, it must cease to output the input, giving up on transparently outputting any valid extensions of the input, so there must not be any valid extensions of the current input. On the other hand, M may temporarily produce an invalid output execution that ends with an action, but only if M is assured that it will be able to output some valid result when the next input result arrives (which we know must eventually occur because the current output ends with an action). Alternatively, when M inputs an invalid execution that ends with an action, it may diverge at some point in

processing its input, giving up on outputting any valid extensions of the action-ending input (so there again must not be any valid extensions of the invalid input). Moreover, for M to enforce \hat{P} on any invalid, natural, infinite-length input execution x , some (finite) prefix of x ending with a result must be invalid (otherwise transparency would force M to output every valid prefix of x that ends with a result, i.e. *invalid* x overall, on input x). Putting all this together, we have requirements on \hat{P} that all invalid, natural, result-ending executions must be dead; a natural action-ending execution x may be invalid but only if it is either dead or can be extended with some result r such that $\hat{P}(x; r)$; and for all invalid, natural, infinite-length executions x , there exists some invalid result-ending prefix of x (i.e., there exists a prefix of x that must be dead). These three constraints imply the result-escape-safety constraint in the theorem statement. ■

Theorem 18. *A property \hat{P} on a system with event set E can be enforced by some $MRA_{exit, \leftrightarrow} M$ if there exist decidable $f : E^* \times \mathbb{N} \rightarrow \dot{A}$ and $g : E^* \rightarrow \dot{R}$ such that:*

1. $aalive_f(\cdot)$ (Empty-aalive)
2. $\forall \varepsilon \in E^* : (alive(\varepsilon) \implies aalive_f(\varepsilon; g(\varepsilon)))$ (Implied-aalive)
3. $\forall \varepsilon \in E^\omega : (\neg \hat{P}(\varepsilon) \implies \exists x \preceq \varepsilon : \neg alive(x))$ (ω -safety)

Proof. We construct an $MRA_{exit, \leftrightarrow} M$ that enforces any such \hat{P} as follows:

- States: $Q = E^* \times E_{\square}^* \times \dot{R} \times \mathbb{N}$ (the input execution up to its death, the output execution, any special result output immediately after the input's death, and the number of elements of \dot{A} output since the input's death)
- Start state: $q_0 = (\cdot, \cdot, \cdot, 0)$ (no events input or output initially)
- Transition function (for simplicity we write δ in terms of high-level transitions):
Consider processing an input action e in state $q = (\varepsilon_i, \varepsilon_o, \rho, j)$.
 - (a) If $e \neq \mathbf{exit}$ and $alive(\varepsilon_o; e)$, then output e and continue in state $(\varepsilon_i; e, \varepsilon_o; e, \cdot, 0)$.
 - (b) Else enter a loop to do the following (while disregarding all additional input events):
 - (i) If $\hat{P}(\varepsilon_o; \mathbf{exit})$ then terminate by outputting the \mathbf{exit} action.
 - (ii) Else if $j=0$, ε_o ends with an action, and $g(\varepsilon_i) \neq \cdot$, output result $g(\varepsilon_i)$ and continue the loop in state $(\varepsilon_i, \varepsilon_o; g(\varepsilon_i), g(\varepsilon_i), 0)$.
 - (iii) Else find the smallest $k \geq j$ such that $f(\varepsilon_i; \rho, k) \neq \cdot$ and output that action $f(\varepsilon_i; \rho, k)$. Then if ε_o ends with an action continue the loop in state $(\varepsilon_i, \varepsilon_o; \square; f(\varepsilon_i; \rho, k), \rho, k+1)$. If ε_o does not end with an action continue the loop in state $(\varepsilon_i, \varepsilon_o; f(\varepsilon_i; \rho, k), \rho, k+1)$.

M maintains an invariant on all high-level states $(\varepsilon_i, \varepsilon_o, \rho, j)$ it reaches that: ε_i is M 's input execution up to the input execution's death (if it does die), ε_o is M 's output execution, ρ is the result $g(\varepsilon_i)$ output immediately after the input execution ε_i dies (if it does die), j is the number of elements of \dot{A} output since the input execution's death, transition (a) will be executed zero or more times followed by zero or more (b) transitions, as long as M is making (a) transitions $alive(\varepsilon_i)$ and M 's target, system, input, and output executions all equal the natural ε_i , when M first makes a (b) transition, it does so from a state in which $aalive_f(\varepsilon_i; g(\varepsilon_i))$ and $\rho = \cdot$ and $j = 0$, M executes transition (b)(ii) at most once, when M first makes a (b)(iii) transition, it does so from a state in which $\varepsilon_o = \varepsilon_i; g(\varepsilon_i)$ (as long as $\varepsilon_i; g(\varepsilon_i)$ is a well-formed execution) and $\rho = g(\varepsilon_i)$ and $j = 0$, ρ can only change in transition (b)(ii), and the entirety of M 's output during (b)(iii) transitions is an execution x such that $actions(x) = f(\varepsilon_i; g(\varepsilon_i), 0); \dots; f(\varepsilon_i; g(\varepsilon_i), j-1)$ and $results(x) = \cdot$ and $\varepsilon_o = \varepsilon_i; g(\varepsilon_i); x$. M establishes this invariant because $q_0 = (\cdot, \cdot, \cdot, 0)$ and if the first action is \mathbf{exit} then we already have $aalive_f(\cdot)$ (by the first constraint in the theorem statement). M maintains the

invariant in every high-level transition it makes (in part because the $alive(\varepsilon; e)$ condition in transition (a) implies $aalive_f(\varepsilon; e; g(\varepsilon; e))$ by the second constraint in the theorem statement, and in part because Lemma 6 implies that as long as M outputs all its input events, its target, system, input, and output executions will be natural and equal).

Consider M 's output on any valid input $\varepsilon_i \in E^\infty$. Because ε_i is valid, all of its prefixes are alive, so M must accept and output verbatim all prefixes of ε_i using only transition (a), until inputting the **exit** action if ε_i is finite, at which point M executes transition (b)(i) to output **exit**. M is therefore transparent. M is also sound. M only terminates with transition (b)(i), which only occurs when its output is valid. M only diverges by infinitely following transition (a) or by entering an infinite loop in transition (b). If M enters an infinite loop in transition (a), all of M 's (infinite number of) input events are alive, so M 's input execution must be valid (because the third constraint in the theorem statement prevents an invalid infinite-length input from having all alive prefixes); therefore, when M loops infinitely with transition (a) it is sound because it outputs every prefix, and only prefixes, of a valid infinite-length input. On the other hand, if M loops infinitely with transition (b), M 's invariant ensures that its entire output is $\varepsilon_i; g(\varepsilon_i); x$ such that $aalive_f(\varepsilon_i; g(\varepsilon_i))$, $actions(x) = f(\varepsilon_i; g(\varepsilon_i), 0); f(\varepsilon_i; g(\varepsilon_i), 1); \dots$, and $results(x) = \cdot$. In this case, the definition of $aalive_f(\varepsilon_i; g(\varepsilon_i))$ implies that $\hat{P}(\varepsilon_i; g(\varepsilon_i); x)$. In all cases, then, M 's output execution is valid, so M is sound and correctly enforces \hat{P} . ■

Theorem 20. *If a property \hat{P} on a system with event set E can be enforced by some $MRA_{\text{exit}, \leftarrow} M$ such that $\forall \varepsilon \in E^* : (alive(\varepsilon) \implies \varepsilon \Downarrow_M \varepsilon)$, then there exist decidable $f : E^* \times \mathbb{N} \rightarrow \dot{A}$ and $g : E^* \rightarrow \dot{R}$ such that:*

1. $aalive_f(\cdot)$ (Empty-aalive)
2. $\forall \varepsilon \in E^* : (alive(\varepsilon) \implies aalive_f(\varepsilon; g(\varepsilon)))$ (Implied-aalive)
3. $\forall \varepsilon \in E^\omega : (\neg \hat{P}(\varepsilon) \implies \exists x \preceq \varepsilon : \neg alive(x))$ (ω -safety)

Proof. Define $g(\varepsilon)$ to be \cdot if $\hat{P}(\varepsilon)$, ε is dead, or ε does not end with an action. When ε is invalid, alive, and ends with an action, iterate through R for a result r such that $alive(\varepsilon; r)$ and let $g(\varepsilon) = r$ (r must exist because $alive(\varepsilon)$ and $\neg \hat{P}(\varepsilon)$, so g is decidable). Note that this definition of g ensures that for all $\varepsilon \in E^*$, if ε is alive then $\varepsilon; g(\varepsilon)$ (1) is valid or (2) is invalid, alive, and does not end with an action.

Define $f(\cdot, j)$ by simulating M on input **exit**, stopping when its output execution is valid or $j+1$ actions have been output. According to MRA semantics and the fact that M enforces \hat{P} , M on input **exit** must output some valid execution x either of the form $a_0; \square; a_1; \square; \dots; a_n$ or of the form $a_0; \square; a_1; \square; \dots$. If M on input **exit** outputs a_j ($j \in \mathbb{N}$) in this action-only execution x , then $f(\cdot, j) = a_j$; otherwise $f(\cdot, j) = \cdot$.

When $\varepsilon \neq \cdot$, define $f(\varepsilon, j)$ as follows. If ε is valid or dead or ends with an action, $f(\varepsilon, j) = \cdot$. Otherwise, when ε is invalid, alive, and does not end with an action, first simulate M on input ε . By the assumption that $\forall \varepsilon \in E^* : (alive(\varepsilon) \implies \varepsilon \Downarrow_M \varepsilon)$, M will output ε verbatim during this simulation. Then continue the simulation by sending M the **exit** action as its next input, and simulate until M 's output execution is valid or $j+1$ actions have been output. According to MRA semantics and the fact that M enforces \hat{P} , M on input **exit** must then output some valid extension x of the current output, either of the form $a_0; \square; a_1; \square; \dots; a_n$ or of the form $a_0; \square; a_1; \square; \dots$. If during this **exit** part of the simulation M outputs a_j ($j \in \mathbb{N}$) in action-only execution x , then $f(\varepsilon, j) = a_j$; otherwise $f(\varepsilon, j) = \cdot$.

Function f is decidable because all of its operations—testing whether finite executions are valid, alive, or end with actions, and simulating M until some event known to be output actually does get output—converge. Also, the construction of f ensures that if ε is \cdot or is invalid, alive, and non-action-ending, then $aalive_f(\varepsilon)$ because there exists an execution $x \in E_\square^\infty$ such that $actions(x) = f(\cdot, 0); f(\cdot, 1); \dots$, $results(x) = \cdot$, and $\hat{P}(x)$ (due to M being sound). Therefore, we have $aalive_f(\cdot)$, satisfying the first obligation in the theorem statement. To satisfy the third obligation in the theorem statement (the implied-aalive constraint), recall from above that for all $\varepsilon \in E^*$, if ε is alive then $\varepsilon; g(\varepsilon)$ (1) is valid or (2) is invalid, alive, and does not

end with an action. In case (1), when $\varepsilon;g(\varepsilon)$ is valid, we trivially have $aalive_f(\varepsilon;g(\varepsilon))$ because any valid execution can be extended with an empty action-only execution to become valid. In case (2), when $\varepsilon;g(\varepsilon)$ is invalid, alive, and does not end with an action, construction of f ensures $aalive_f(\varepsilon;g(\varepsilon))$, as noted above.

Only the third constraint (ω -safety) in the theorem statement remains to be proved. If this constraint did not hold then there would exist some invalid infinite execution x whose prefixes are all alive. Using the assumption that $\forall \varepsilon \in E^* : (alive(\varepsilon) \implies \varepsilon \Downarrow_M \varepsilon)$, then, M must output every prefix of x verbatim when that prefix is input, so M 's overall behavior on input x will be to output all prefixes, and only prefixes, of x (i.e., $x \Downarrow_M x$). This behavior contradicts the fact that M enforces \hat{P} because M cannot be sound while outputting an invalid execution. Therefore, \hat{P} must be an ω -safety property. ■

Theorem 21. *If a property \hat{P} on a system with event set E can be enforced by some $MRA_{exit,\leftarrow} M$ then there exists decidable $f : E^* \times \mathbb{N} \rightarrow \dot{A}$ such that $aalive_f(\cdot)$.*

Proof. Define $f(\cdot, j)$ by simulating M on input **exit**, stopping when its output execution is valid or $j+1$ actions have been output. According to MRA semantics and the fact that M enforces \hat{P} , M on input **exit** must output some valid execution x either of the form $a_0; \square; a_1; \square; \dots; a_n$ or of the form $a_0; \square; a_1; \square; \dots$. If M on input **exit** outputs a_j ($j \in \mathbb{N}$) in this action-only execution x , then $f(\cdot, j) = a_j$; otherwise $f(\cdot, j) = \cdot$. Function f also returns \cdot on all other inputs. This function is decidable because $j \in \mathbb{N}$ and M must output some valid sequence of actions on input **exit**. Moreover, $aalive_f(\cdot)$ because $actions(x) = f(\cdot, 0); f(\cdot, 1); \dots$, $results(x) = \cdot$, and $\hat{P}(x)$. ■

Theorem 22. *There exist $MRA_{exit,\leftarrow} M$ and property \hat{P} on a system with event set E such that M enforces \hat{P} but \hat{P} is not an implied-alive property—that is, $\exists \varepsilon \in E^* : (alive(\varepsilon) \wedge \neg aalive_f(\varepsilon;g(\varepsilon)))$, for all decidable $f : E^* \times \mathbb{N} \rightarrow \dot{A}$ and $g : E^* \rightarrow \dot{R}$.*

Proof. Let $A = \{a_1, a_2, \mathbf{exit}\}$, $R = \{r_1, r_2\}$, $E = A \cup R$, and for all $x \in E_{\square}^{\infty}$, $\hat{P}(x)$ if and only if $x = \mathbf{exit}$, $x = \square; r_1; \mathbf{exit}$, or $x = a_1; r_1; a_2; r_2; \mathbf{exit}$. This \hat{P} is not an implied-alive property because natural execution a_1 (and similarly for $a_1; r_1$) is alive but there is no way to extend a_1 with a $\rho \in \dot{R}$ (computed by g) and then an action-only execution x (computed by f) such that $\hat{P}(a_1; \rho; x)$.

An $MRA_{exit,\leftarrow} M$ can enforce this \hat{P} by doing the following:

- If the first action input is **exit** or a_2 , simply output **exit**.
- If the first action input is a_1 then output r_1 .
 - Then if the next action input is not a_2 , output **exit** (for a total output of $\square; r_1; \mathbf{exit}$).
 - Otherwise, the next action input is a_2 , so output a_1 , input a result for a_1 , output a_2 , input a result for a_2 , output r_2 , input the next action, and output **exit**. In this case, the overall target execution is $a_1; r_1; a_2; r_2; a$ (for some $a \in A$), and the overall system execution is $a_1; r; a_2; r'; \mathbf{exit}$ (for some $r, r' \in R$). By the definition of $output_*$ in Figure 4, the total output execution in this case is $x = a_1; r_1; a_2; r_2; \mathbf{exit}$.

In all cases, M 's output satisfies \hat{P} and **exit** \Downarrow_M **exit** and $a_1; r_1; a_2; r_2; \mathbf{exit}$ \Downarrow_M $a_1; r_1; a_2; r_2; \mathbf{exit}$, so M is sound and transparent with respect to \hat{P} . ■

Theorem 23. *There exist $MRA_{exit,\leftarrow} M$ and property \hat{P} on a system with event set E such that M enforces \hat{P} but \hat{P} is not an ω -safety property—that is, $\exists \varepsilon \in E^{\omega} : (\neg \hat{P}(\varepsilon) \wedge \forall x \preceq \varepsilon : alive(x))$.*

Proof. Let $E = A \cup R$, where A and R are any disjoint sets such that $a \in A$, $\mathbf{exit} \in A$, and $r \in R$. Define $\hat{P}(x)$ to be true for $x \in E_{\square}^{\infty}$ if and only if x matches one of the patterns $(a; r)^* \mathbf{exit}$ or $(\square; r)^* \mathbf{exit}$ or $(\square; r)^{\omega}$. This \hat{P} is not an ω -safety property because execution $a; r; a; r; \dots$ is invalid despite all of its prefixes being alive.

An $\text{MRA}_{\mathbf{exit}, \leftarrow} M$ can enforce this \hat{P} by starting a *counter* at zero and entering a loop to do the following:

- If the input action is a , increment *counter* and output r .
- If the input action is \mathbf{exit} , output *counter* a 's (inputting some result for each a output) and then terminate by outputting \mathbf{exit} .
- Otherwise, terminate by outputting \mathbf{exit} .

M is transparent because if it inputs n repetitions of alternating a 's and r 's followed by \mathbf{exit} , it outputs exactly that input execution (one a and r for every a input). M is also sound because its output execution will always match $(a; r)^* \mathbf{exit}$ or $(\square; r)^* \mathbf{exit}$ or $(\square; r)^{\omega}$, all of which satisfy \hat{P} . ■