2009

Program monitoring in a mandatory-results model

Srikar Reddy Reddy
University of South Florida

Program Monitoring in a Mandatory-results Model

by

Srikar Reddy Reddy

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Science
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Date of Approval:
July 1, 2009

Keywords: Security policies, Safety, Liveness, Security automata, Policy enforcement

ACKNOWLEDGEMENTS

I am extremely thankful to my advisor Dr. Jay Ligatti, for his guidance and motivation thoughout the completion of this thesis. This thesis would not have existed in its current form unless for his never ending support. This thesis derives from the current working project titled ”Security Automata with Mandatory Results”. I would like to thank Dr. Larry Hall and Dr. Rahul Tripathi for being on my thesis supervisory committee.

I also like to thank my lab mates at the security and programming languages lab for their valuable suggestions in formatting this thesis. Last but not least I like to thank my friends at University of South Florida for their moral support.

This research was supported in part by NSF grants CNS-0716343 and CNS-0742736.

TABLE OF CONTENTS

LIST OF FIGURES

Program Monitoring in a Mandatory-results Model

Srikar Reddy Reddy

ABSTRACT

In many real enforcement systems, a security-relevant action must return a result before the application program that invoked that action can continue to execute. However, current models of runtime mechanisms do not capture this requirement on results being returned to application programs; current models are limited to reasoning about policies and enforcement in terms of actions alone, without considering the results of those actions. This thesis presents a more general model of runtime policy enforcement in which all actions return (possibly void- or unit-type) results. This mandatory-results model more accurately reflects the capabilities and limitations of real enforcement mechanisms, particularly those mechanisms that operate by monitoring function/method invocations. We analyze the new model to show that result-returning runtime monitors enforce a strict superset of the safety policies, including some nontrivial liveness policies.

CHAPTER 1

INTRODUCTION

Runtime enforcement mechanisms dynamically monitor untrusted target applications to ensure that those applications adhere to desired policies. Although computability and proof theory provide good frameworks for understanding which policies can be enforced on target applications through static analysis, we currently lack sufficiently general frameworks for understanding which policies can be enforced through runtime mechanisms. This thesis 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 mechanisms (in firewalls, operating systems, auditing tools, spam filters, intrustion-detection systems, access-control systems, etc).

Much research has been performed to model generic runtime mechanisms and analyze their enforcement capabilities (cf. Chapter 5.1). However, existing models of generic runtime mechanisms are based on the system abstractions shown in Figure 1.1. Figure 1.1a 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 1.1b. 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).

Although existing models of software security fit into the framework of Figure 1.1, 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

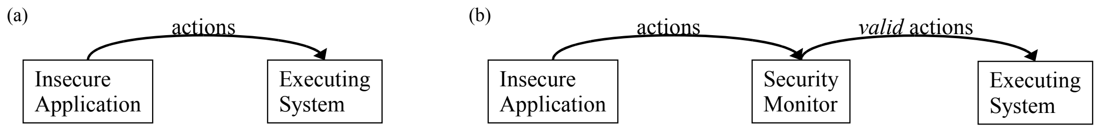

Figure 1.1. Existing model of an insecure application (a) secured by a program monitor (b).

result for a. Figure 1.2a depicts this more practical scenario, which encompasses application actions that:

1. Return results, such as actions for dereferencing pointers, for returning user input, for reading and returning data in a file or network buffer, etc.

2. Raise exceptions, such as actions for dereferencing (possibly null) pointers, writing to (possibly nonexistent) files or network ports, etc. In this case, we simply treat the exceptions that can be raised as potential return values.

3. 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 1.2b illustrates. This is a crucial capability for enforcing many security policies, such as privacy, access-control, and information-flow policies, which may require mechanisms to sanitize the results of actions before applications access those results. For example, policies may require that system files get hidden when user-level applications retrieve directory listings, or that email messages flagged by spam filters do not get returned to email-client applications. Because existing frameworks for reasoning about generic runtime mechanisms do not model results of actions, one cannot use existing frameworks to specify or reason about enforcement of such policies.

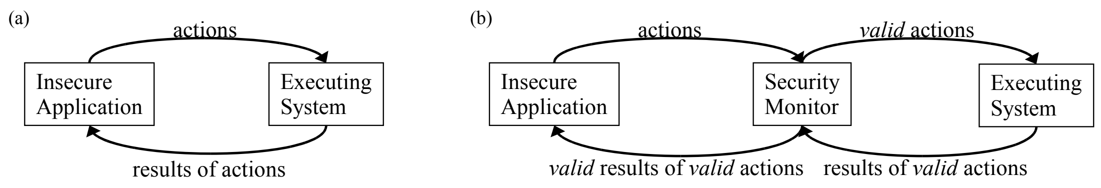

Figure 1.2. This thesis's model of an insecure application with results (a) secured by a program monitor (b).

Contributions This thesis generalizes existing frameworks of runtime enforcement to capture the practical ability of monitors to transform both actions and results, as illustrated in Figure 1.2b. We make the following contributions.

1. Chapter 2 modifies existing definitions of policies, properties, and program executions, to take into account action results.
2. Chapter 3 defines *mandatory-results automata* (MRAs), new models of run-time enforcement mechanisms that interpose on and ensure 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.
3. Chapter 4 analyzes MRAs to show that they enforce a strict superset of the safety policies, including some nontrivial liveness policies.

This thesis 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 here focus 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.

CHAPTER 2

BACKGROUND NOTATION AND DEFINITIONS

Before defining mandatory-results automata (MRAs) and what it means for an MRA to enforce a policy, we first need to define policies and set up some basic notation for specifying systems and traces. Most of the notation and definitions presented in this chapter are extended versions of notation and definitions in previous work (extended to include results of actions) [1, 2, 3].

2.1 Notation

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., Java 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; $E = A \cup R$. Given a set of events E, $act(E)$ refers to all the actions in E and $res(E)$ to all the results in E.

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 executions under consideration in this thesis have the form $a_0; r_0; a_1; r_1; \ldots; a_n; r_n$ or $a_0; r_0; a_1; r_1; \ldots; a_n$ or $a_1; r_1; a_2; r_2; \ldots$ (where r_0 is the result of action a_0, r_1 is the result of a_1, etc.). The symbol $\cdot$ represents a zero-length execution (in which no events occur). When event e occurs in execution x, we write $e \in x$. We denote the sequence of actions in an execution x as $acts(x)$ and the sequence of results in x as $rslts(x)$. Moreover, we denote the i^{th} event ($i \in \mathbb{N}$) of execution x as $x[i]$ and the length of finite execution x as $|x|$. We also 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 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 $\cdot$.

The notation $x;\varepsilon$ represents concatenation of two executions x and ε (the first of which must have finite length). When x is a finite prefix of ε we write $x \preceq \varepsilon$ or $\varepsilon \succeq x$, and when x is a proper prefix of ε we write $x \prec \varepsilon$ or $\varepsilon \succ x$. Finally, when E is clear from context, we make extensive use of abbreviations of the form $\exists x \preceq \varepsilon : F$ in place of the more verbose $\exists x \in E^* : (x \preceq \varepsilon \ \wedge \ F)$.

2.2 Policies and Properties

A *policy* is a predicate on sets of executions [2]; a set of executions $\mathcal{X} \subseteq E^\infty$ 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^∞ such that for all $\mathcal{X} \subseteq E^\infty$, the following is true [2]:

$$P(\mathcal{X}) \Longleftrightarrow \forall x \in \mathcal{X} : \hat{P}(x) \qquad \text{(Property)}$$

This distinction between properties and more general policies is important when reasoning about dynamic enforcement mechanisms because such 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 all executions of an application.

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. When $\hat{P}(\varepsilon)$, we say that ε *satisfies* or *obeys* the property, or ε is *valid* or *legal*; similarly, if $\neg\hat{P}(\varepsilon)$ then we say that ε *violates* or *disobeys* the property, or ε is *invalid* or *illegal*. Properties satisfied by the empty execution that are decidable over finite-length executions are called *reasonable* properties.

Because the definitions of policy and property above operate on executions containing results, it is possible to define policies that take results into consideration. For example, a policy might be satisfied by exactly those sets of executions $\mathcal{X}$ in which all natural numbers n get returned as the result of action a in some execution in $\mathcal{X}$. 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 a 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 `ls` action (in any execution in $\mathcal{X}$) contains the string `.hidden`. This policy is a property because it is satisfied if and only if every execution in $\mathcal{X}$ lacks `.hidden`-containing results to all `ls` actions.

Properties, such as the hidden-files policy just described, specifying that "nothing bad ever occurs" are called *safety* properties [4]. This well-studied class of properties includes commonly enforced properties such as access-control properties (specifying that no illegal accessing of resources ever occurs). Technically, safety means that every invalid execution must have some invalid prefix after which all extensions are invalid [5], so a property $\hat{P}$ on a system with event set E is a safety property if and only if:

$$\forall \varepsilon \in E^{\infty} : (\neg\hat{P}(\varepsilon) \Longrightarrow \exists \varepsilon' \preceq \varepsilon : \forall x \succeq \varepsilon' : \neg\hat{P}(x)) \qquad \text{(SAFETY)}$$

On the other hand, *liveness* properties state that nothing irremediably bad ever happens in a finite amount of time [1]. 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:

$$\forall \varepsilon \in E^* : \exists x \succeq \varepsilon : \hat{P}(x) \tag{\textsc{Liveness}}$$

Exactly one policy is both a safety and a liveness property: the property $\top$, which considers all executions valid [1].

CHAPTER 3

PROPERTY ENFORCEMENT WITH MANDATORY-RESULTS AUTOMATA (MRAS)

Having defined executions, policies, and properties in the previous chapter, we are ready to formally model monitors that behave as in Figure 1.2b.

3.1 Operational Semantics of MRAs

Synchronous application actions impose a major constraint on how monitors may behave; the monitors must return a result to the target before seeing the next action the target wishes to execute. Real program monitors (e.g., as created in the Naccio [6], PoET/PSLang [7], Polymer [8], LoPSiL [9], and ConSpec [10] systems) have this mandatory-results constraint, and it is for this constraint that mandatory-results automata are named.

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 transition function of the form $\delta : Q \times E \to Q \times E$, which takes M's current state and an event being input to M (either an action the target is attempting to execute or a result the underlying system has produced) and returns a new state for M and an event to be output from M (either an action to be executed on the underlying system or a result to be returned to the target). In contrast to previous work [11, 12, 3], we do not require δ to be decidable (it may not halt on some inputs); this ability of MRAs to diverge accurately models the abilities of real runtime mechanisms.

We call ${}^{\alpha_i}_{\rho_o}\Big[q\Big]^{\alpha_o}_{\rho_i}$ a *configuration* of monitor 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 ${}^{\alpha_i}_{\rho_o}\Big[q\Big]^{\alpha_o}_{\rho_i}$, we simply write the configuration as q. Also, we normally do not bother to write the dots in configurations, so ${}^{a}\Big[q\Big]_{r}$ is the same as ${}^{a}_{\cdot}\Big[q\Big]^{\cdot}_{r}$. The starting configuration of an MRA is ${}^{a}\Big[q_0\Big]$; that is, the monitor begins executing in its initial state with the first application action being input. Notice that our bracket-based notation for configurations matches the graphic representation of monitors' inputs and outputs as shown in Figure 1.2b.

We are now ready to describe the operational semantics of MRAs, as defined by a labeled single-step judgment whose form is ${}^{\alpha_i}_{\rho_o}\Big[q\Big]^{\alpha_o}_{\rho_i} \xrightarrow[\varepsilon_S]{\varepsilon_T} M \; {}^{\alpha_i'}_{\rho_o'}\Big[q'\Big]^{\alpha_o'}_{\rho_i'}$. This judgment means that MRA M is taking a single step from configuration ${}^{\alpha_i}_{\rho_o}\Big[q\Big]^{\alpha_o}_{\rho_i}$ to configuration ${}^{\alpha_i'}_{\rho_o'}\Big[q'\Big]^{\alpha_o'}_{\rho_i'}$ 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 used in our single-step judgment: because M will always be clear from the context, we normally omit it from the judgment.

The definition of MRAs' single-step semantics appears in Figure 3.1. There are three inference rules defining MRA transitions in Figure 3.1; each rule allows an MRA to transition from a current state q to a new state q'. The first rule (*Ins-on-Act*) enables an MRA to insert an action a' that will be executed before the monitor returns a result to the target for the action a currently being input to the MRA; because a' will be executed, it is placed into the system execution. The second rule (*Res-on-Ins*) enables an MRA to update its state in response to receiving the result of an action it just inserted; because the result is coming from the underlying system, it too gets placed into the system execution. The third rule (*Res-on-Act*) enables an MRA to return a result r to the target application for an action a that the target has sent as input to the MRA; because a and r are events viewable

$$\boxed{{}^{\alpha_i}_{\rho_o}\left[q\right]^{\alpha_o}_{\rho_i} \xrightarrow[\varepsilon_S]{\varepsilon_T} {}^{\alpha_i'}_{\rho_o'}\left[q'\right]^{\alpha_o'}_{\rho_i'}}$$

$$\frac{\delta(q,a)=(q',a')}{{}^{a}\left[q\right] \xrightarrow[a']{} {}^{a}\left[q'\right]^{a'}}\ \textit{(Ins-on-Act)}$$

$$\frac{\delta(q,r)=(q',\_)}{{}^{a}\left[q\right]_{r} \xrightarrow[r]{} {}^{a}\left[q'\right]}\ \textit{(Res-on-Ins)}$$

$$\frac{\delta(q,a)=(q',r)}{{}^{a}\left[q\right] \xrightarrow{a;r} {}_{r}\left[q'\right]}\ \textit{(Res-on-Act)}$$

Figure 3.1. Single-step semantics of mandatory-results automata (monitor-controlled transitions).

by the target, they get placed into the target execution. The simplicity of these inference rules, as much as is present, is the product of much effort and iterating through numerous less-satisfactory attempts.

In addition to the monitor-controlled transitions shown in Figure 3.1, there are two single-step transitions between configurations that can occur outside of the monitor's control. First, the transition ${}_{r}\left[q\right] \longrightarrow {}^{a}\left[q\right]$ occurs when the monitor has finished processing an input action by returning a result r to the target, and then the target sends its next action a to the monitor; however, if the target generates no additional actions then the MRA never makes a transition from (and therefore terminates in) configuration ${}_{r}\left[q\right]$. The second single-step transition outside of the monitor's control is ${}^{a}\left[q\right]^{a'} \rightarrow {}^{a}\left[q\right]_{r}$, which occurs when the executing system returns a result r for an inserted action a'. This transition always occurs in our model from configuration ${}^{a}\left[q\right]^{a'}$ because we assume that the underlying system returns a result for every action it is asked to execute.

We next make several observations about the operational semantics of MRAs:

1. A wildcard ($\_$) exists in the premise of rule *Res-on-Ins* in place of an event e because the rule, which causes no event to be output from the MRA, ignores e's value.

2. An MRA can "accept" an application action a by inserting it (with rule *Ins-on-Act*), receiving and remembering the result r of executing a (with rule *Res-on-Ins*), and then returning r to the application (with rule *Res-on-Act*).

3. An MRA can "halt" an application by inserting an action like `exit`, if the underlying system can execute such an action. Alternatively, an MRA can enter an infinite loop to block additional actions and results from being input and output. An MRA can achieve a similar effect by repeatedly transitioning with the *Res-on-Act* rule, which will cause the system (but not target) execution to terminate.

4. MRAs are 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 action by returning a result for it.

5. We make no assumptions about the underlying system that executes application actions beyond that it produces some result (possibly an exception) for every action it is asked to execute. The underlying 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 underlying system execute those actions. This design also makes the single-step relation nondeterministic, as the system may return any of a number of results for the same action in an execution.

6. Just as MRAs are agnostic of how the underlying system generates results, MRAs are agnostic of 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.

These observations describe ideas that match our understanding of how real program monitors (such as are implemented in the Naccio [6], PoET/PSLang [7], Polymer [8], LoPSiL [9], and ConSpec [10] systems) behave.

Having defined the single-step semantics of MRAs, we define the multi-step judgment in the standard way as the reflexive, transitive closure of the single-step judgment. The multi-step judgment form is $q \xrightarrow[\varepsilon_S]{\varepsilon_T}{}^{*} q'$ where ε_T and ε_S are the target and system executions formed (via concatenations) during the multi-step transition from configuration q to configuration q'.

3.2 MRA-based Enforcement

We adopt the notion that mechanisms enforce policies when they are *sound* and *transparent* [13, 14, 15, 3]. Soundness requires that the *observable* execution, which is the execution *output* from the mechanism, must satisfy the desired policy. Soundness alone is unsatisfactory, though, because any mechanism can enforce any reasonable property by simply rejecting all inputs and always outputting an empty execution (thus, all of its observable outputs are valid). Transparency prevents this situation by requiring mechanisms to preserve the semantics of valid *input* executions; if the application and system only provide valid inputs to the mechanism, then the mechanism must not alter the semantics of those valid inputs.

To reason about transparency and whether a mechanism preserves the semantics of valid input, we assume the presence of a system-specific equivalence relation $\approx$ on (possibly infinite-length) executions. When $x \approx x'$ we say that x and x' are *semantically equivalent.* We place no constraints on this equivalence relation beyond that properties may not distinguish between equivalent executions.

One more judgment will make it easier to define enforcement. For reasoning about both soundness and transparency, we need a way to say that an enforcement mechanism M takes ε_i as its input, and *transforms* ε_i into the output execution ε_o. This idea of viewing the role of enforcement mechanisms as transforming possibly invalid input executions into valid executions (soundness), while preserving the semantics of valid input executions (transparency), comes from earlier work [11, 3]. When mechanism M transforms input execution ε_i into the output execution ε_o, we write $\varepsilon_i \Downarrow_M \varepsilon_o$. Defining the $\Downarrow_M$ judgment for MRAs is a bit tricky, so we postpone that definition until we finish defining enforcement. For now, and throughout Chapter 3.2.1, let us assume that we already have a definition of $\Downarrow_M$ for all M that are MRAs.

3.2.1 Effective Enforcement

We define *enforcement* in terms of soundness and transparency. As in earlier work [11, 3], we call this style of enforcement "effective$_{\approx}$ enforcement" because it defines when a mechanism is behaving fully effectively and relies on the equivalence relation $\approx$.

Definition 1 (Effective$_{\approx}$ Enforcement)**.** *An MRA $M=(E,Q,q_o,\delta)$* effectively$_{\approx}$ enforces *a property $\hat{P}$ on a system with event set E and semantic equivalence relation $\approx$ if and only if:*

$$\forall \varepsilon_i, \varepsilon_o \in E^{\infty} : (\varepsilon_i \Downarrow_M \varepsilon_o) \implies (\hat{P}(\varepsilon_o) \wedge (\hat{P}(\varepsilon_i) \implies \varepsilon_i \approx \varepsilon_o))$$

Because equivalence relations are system specific, and because earlier work has proved that equivalence relations enable security automata to enforce *any* reasonable property (including, for example, termination) [3], we follow previous work and focus here on systems in which the equivalence relation $\approx$ is just the (system-independent) equality relation $=$. By focusing on execution equality, we are considering lower bounds on true enforcement capabilities; any property effectively$_{=}$ enforceable on a system with equivalence relation $\approx$ must also be effectively$_{\approx}$ enforceable (because equality implies equivalence).

To reason about possibly infinite-length executions being equal, we formally define two executions to be equal when they share all the same prefixes.

Definition 2 (Equality of Executions)**.** *For all event sets E and executions $x_1, x_2 \in E^{\infty}$, $x_1 = x_2$ if and only if $(\forall \varepsilon \preceq x_1 : \varepsilon \preceq x_2)$ and $(\forall \varepsilon \preceq x_2 : \varepsilon \preceq x_1)$.*

For the remainder of this thesis, we will write "effective enforcement" or just "enforcement" in place of "effective$_=$ enforcement".

3.2.2 Input and Output Executions with MRAs

We now return to the postponed issue of which executions MRAs input and output; our goal is to define when the $\varepsilon_i \Downarrow_M \varepsilon_o$ judgment holds for an MRA M and possibly infinite-length executions ε_T and ε_S. There are several hurdles to overcome in order to define this transforms relation, but its definition will enable us to judge whether given MRAs enforce desired properties.

We have already defined (in Chapter 3.1) how MRAs build up target and system executions with the multi-step judgment $q \xrightarrow[\varepsilon_S]{\varepsilon_T}{}^{*} q'$. However, this judgment only lets us reason about the target and system executions after a finite number of steps. MRAs may make an infinite number of transitions while monitoring a program run, so we must generalize the multi-step judgment to handle possibly infinite-length target and system executions built up through possibly infinitely many MRA transitions. Hence, we introduce the judgment $\varepsilon_T \xleftrightarrow[M]{} \varepsilon_S$ to indicate that the MRA M, beginning in its initial state, builds target and system executions ε_T and ε_S during its entire run. Intuitively, $\varepsilon_T \xleftrightarrow[M]{} \varepsilon_S$ holds if and only if, an any finite number of steps, M only builds target and system executions that are prefixes of ε_T and ε_S, and conversely, M builds target and system executions containing every prefix of ε_T and ε_S. Formally, we define $\varepsilon_T \xleftrightarrow[M]{} \varepsilon_S$ as follows.

Definition 3 (Possibly Infinite-length Target and System Executions)**.** *For all MRA* $M = (E, Q, q_o, \delta)$ *and* $\varepsilon_T, \varepsilon_S \in E^\infty$, $\varepsilon_T \xleftrightarrow[M]{} \varepsilon_S$ *if and only if:*

1. $\forall q' \in Q : \forall \varepsilon'_T \in E^* : \forall \varepsilon'_S \in E^* : ((\, q_o \xrightarrow[\varepsilon'_S]{\varepsilon'_T}{}^{*} q' \;\wedge\; acts(\varepsilon'_T) \preceq acts(\varepsilon_T) \;\wedge\; rslts(\varepsilon'_S) \preceq rslts(\varepsilon_S)) \implies (\varepsilon'_T \preceq \varepsilon_T \wedge \varepsilon'_S \preceq \varepsilon_S))$

2. $\forall \varepsilon'_T \preceq \varepsilon_T : \forall \varepsilon'_S \preceq \varepsilon_S : \exists q' \in Q : \exists \varepsilon''_T \succeq \varepsilon'_T : \exists \varepsilon''_S \succeq \varepsilon'_S : \; q_o \xrightarrow[\varepsilon''_S]{\varepsilon''_T}{}^{*} q'$

To define the transforms judgment, we also must define which execution a monitor inputs and which it outputs. Until now, our rules for MRAs have built target and system executions, rather than input and output executions, so we have to define how to convert target and system executions into input and output executions. Target and system executions are the executions visible to targets and systems, but input and output executions are the executions input to, and output from, the MRA itself. That is, an input execution includes the actions input from the target and the results of those actions as input from the underlying system. An output execution includes actions output to the system and the results of those actions as output to the target program. Hence, soundness will require that the actions actually executed by the system and the results actually returned to the target are valid, allowing us to reason about MRAs enforcing result-sanitization properties, such as the hidden-files policy described in Chapter 2.2. Also, transparency will require that if the MRA receives a valid stream of actions from the target and results from the system, then it must end up outputting those same actions to the system and those same results to the target.

Given possibly infinite-length target and system executions ε_T and ε_S, we build the input execution by taking all the actions in ε_T and interspersing the results to those actions as first found in ε_S. Similarly, we build the output execution 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, an issue arises when an action a in ε_T is not present in ε_S, so we have no result to include for a in ε_i. In this case we keep a in ε_i but use a's result r from ε_T, rather than from ε_S, as a's result in ε_i. Thus, action a, which the target sent to the MRA, correctly counts as an input action, and because the MRA never actually input a result for a from the system, we treat the result r that the MRA returned to the target for a (which is the only result we have for a) as the input execution's result for a. Note that including r in the input execution does not damage the output execution because the MRA never output a, so r could not be included in the output execution anyway. An analogous issue arises when an

$$\boxed{\mathit{input}(\varepsilon_T, \varepsilon_S) = \varepsilon_i}$$

$$\frac{}{\mathit{input}(\cdot, \varepsilon_S) = \cdot}\ \textit{(Input-Dot)}$$

$$\frac{a \notin \varepsilon_S}{\mathit{input}(a; r; \varepsilon_T,\ \varepsilon_S) = a; r; \mathit{input}(\varepsilon_T, \varepsilon_S)}\ \textit{(Input-No-Swap)}$$

$$\frac{a \notin \varepsilon_S}{\mathit{input}(a; r; \varepsilon_T,\ \varepsilon_S; a; r'; \varepsilon'_S) = a; r'; \mathit{input}(\varepsilon_T,\ \varepsilon_S; \varepsilon'_S)}\ \textit{(Input-Swap)}$$

$$\boxed{\mathit{output}(\varepsilon_T, \varepsilon_S) = \varepsilon_o}$$

$$\frac{}{\mathit{output}(\varepsilon_T, \cdot) = \cdot}\ \textit{(Output-Dot)}$$

$$\frac{a \notin \varepsilon_T}{\mathit{output}(\varepsilon_T,\ a; r; \varepsilon_S) = a; r; \mathit{output}(\varepsilon_T, \varepsilon_S)}\ \textit{(Output-No-Swap)}$$

$$\frac{a \notin \varepsilon_T}{\mathit{output}(\varepsilon_T; a; r'; \varepsilon'_T,\ a; r; \varepsilon_S) = a; r'; \mathit{output}(\varepsilon_T; \varepsilon'_T,\ \varepsilon_S)}\ \textit{(Output-Swap)}$$

Figure 3.2. Definitions of *input* and *output* judgments.

action a in ε_S is not present in ε_T; the issue gets resolved in the same way by keeping a in ε_o and using a's result from ε_S as a's result in ε_o. Putting all these rules together, we would convert target execution $a_1; r_1; a_1; r_1; a_2; r_2; a_3; r_3$ and system execution $a_4; r_4; a_3; r_1; a_1; r_2$ into input execution $a_1; r_2; a_1; r_1; a_2; r_2; a_3; r_1$ and output execution $a_4; r_4; a_3; r_3; a_1; r_1$.

Figure 3.2 defines two judgments ($\mathit{input}(\varepsilon_T, \varepsilon_S){=}\varepsilon_i$ and $\mathit{output}(\varepsilon_T, \varepsilon_S){=}\varepsilon_o$) that hold when *finite-length* target and system executions ε_T and ε_S have been properly converted into *finite-length* input and output executions ε_i and ε_o in the manner just described. We generalize these definitions to infinite-length executions using a technique similar to our generalization of the multi-step judgment (with finite-length ε_T and ε_S) to the $\varepsilon_T \xleftrightarrow[M]{} \varepsilon_S$ judgment (which allows infinite-length ε_T and ε_S). The first generalized judgment, allowing all executions to have infinite length and indicating that target and system executions ε_T and ε_S form input execution ε_i, is written as $\varepsilon_T \otimes_{in} \varepsilon_S = \varepsilon_i$. That is, $\otimes_{in}$ is an operator that builds ε_i simply by starting with ε_T and swapping in results from ε_S for the actions in ε_T.

Similarly, $\varepsilon_T \otimes_{out} \varepsilon_S = \varepsilon_o$ indicates that possibly infinite-length ε_o contains all the actions of ε_S, with the results of those actions in ε_T swapped in. To define the $\varepsilon_T \otimes_{in} \varepsilon_S = \varepsilon_i$ judgment, we intuitively require the *input* function to return only prefixes of ε_i when given prefixes of ε_T, and conversely, the *input* function must return all prefixes of ε_i. The $\varepsilon_T \otimes_{out} \varepsilon_S = \varepsilon_o$ judgment is defined similarly. Formal definitions of the $\otimes_{in}$ and $\otimes_{out}$ operators appear below.

Definition 4 ($\otimes_{in}$)**.** *For all event sets E and $\varepsilon_T, \varepsilon_S, \varepsilon_i \in E^\infty$, $\varepsilon_T \otimes_{in} \varepsilon_S = \varepsilon_i$ if and only if:*

1. $\forall \varepsilon'_T \preceq \varepsilon_T : \exists \varepsilon'_S \preceq \varepsilon_S : \mathit{input}(\varepsilon'_T, \varepsilon'_S) \preceq \varepsilon_i$

2. $\forall \varepsilon'_i \preceq \varepsilon_i : \exists \varepsilon'_T \preceq \varepsilon_T : \exists \varepsilon'_S \preceq \varepsilon_S : \mathit{input}(\varepsilon'_T, \varepsilon'_S) = \varepsilon'_i$

Definition 5 ($\otimes_{out}$)**.** *For all event sets E and $\varepsilon_T, \varepsilon_S, \varepsilon_o \in E^\infty$, $\varepsilon_T \otimes_{out} \varepsilon_S = \varepsilon_o$ if and only if:*

1. $\forall \varepsilon'_S \preceq \varepsilon_S : \exists \varepsilon'_T \preceq \varepsilon_T : \mathit{output}(\varepsilon'_T, \varepsilon'_S) \preceq \varepsilon_o$

2. $\forall \varepsilon'_o \preceq \varepsilon_o : \exists \varepsilon'_S \preceq \varepsilon_S : \exists \varepsilon'_T \preceq \varepsilon_T : \mathit{output}(\varepsilon'_T, \varepsilon'_S) = \varepsilon'_o$

At last, we are ready to define transformation of a possibly infinite-length input execution ε_i to 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 target and system executions ε_T and ε_S, and $\varepsilon_T \otimes_{in} \varepsilon_S = \varepsilon_i$ and $\varepsilon_T \otimes_{out} \varepsilon_S = \varepsilon_o$.

Definition 6 (MRA Transformation)**.** *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 $\exists \varepsilon_T, \varepsilon_S \in E^\infty$: $(\varepsilon_T \xleftrightarrow[M]{} \varepsilon_S \;\wedge\; \varepsilon_T \otimes_{in} \varepsilon_S = \varepsilon_i \;\wedge\; \varepsilon_T \otimes_{out} \varepsilon_S = \varepsilon_o)$.*

CHAPTER 4

ANALYSIS OF MRA-ENFORCEABLE POLICIES

This chapter compares the set of properties MRAs can enforce with the more established sets of safety and liveness properties.

First we show that MRAs can enforce *any* reasonable safety property. The technique for enforcing any safety property with an MRA, as with other security automata [3], is to accept all events as long as they satisfy the safety property and cease execution[1] as soon as an invalid input event is detected. In this way, the MRA will only output valid executions (soundness), 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 all valid executions must also be valid).

Theorem 7. *Any reasonably safety property $\hat{P}$ on a system with event set E can be enforced by some MRA.*

Proof. We construct an MRA M that enforces any such $\hat{P}$ as follows:

1. States: $Q = E^* \times E^*$ (the current target execution paired with the current system execution)

2. Start state: $q_0 = (\cdot, \cdot)$

3. Transition function (for simplicity we write δ in terms of high-level transitions):

   Consider processing an input action a in state $q = (\varepsilon_T, \varepsilon_S)$:

[1]Our proofs in this chapter "cease execution" by having the MRAs enter infinite loops (to block the MRAs from receiving additional inputs and outputs). In practice, runtime mechanisms could enter infinite loops but would typically cease execution more straightforwardly by invoking an `exit` action. We use the infinite-loop approach to cease execution in our proofs because it does not constrain us to consider systems with `exit`-style actions.

(a) If $\hat{P}(\varepsilon_T; a)$, then insert a and let r be the result obtained for a from the system. Then, if $\hat{P}(\varepsilon_T; a; r)$, return r to the target and continue in state $(\varepsilon_T; a; r, \varepsilon_S; a; r)$. Otherwise, if $\neg\hat{P}(\varepsilon_T; a; r)$ then enter an infinite loop.

(b) Otherwise, if $\neg\hat{P}(\varepsilon_T; a)$ then enter an infinite loop.

Now we let $\varepsilon \in E^\infty$ be the automaton input. If $\hat{P}(\varepsilon)$ then by the definition of safety, all prefixes of ε must be valid, so by the definition of M, M will output all the prefixes of ε. Hence, M outputs ε in this case and correctly enforces $\hat{P}$ on valid executions. In the case where $\neg\hat{P}(\varepsilon)$, we must only show that M is sound. By the definition of M, only valid actions and results get output, so M is sound and correctly enforces $\hat{P}$ on all executions. □

Moreover, there exist nonsafety, nonliveness properties enforceable by MRAs.

Theorem 8. *There exists a reasonable property $\hat{P}$ on a system with event set E that is neither a safety nor a liveness property but can be enforced by an MRA M.*

Proof. Let $E = \{a_1, a_2, r\}$, where a_1 and a_2 are actions and r is a result. Also let $\hat{P}$ be satisfied by exactly those executions matching the pattern $(a_1; r)^\infty$. $\hat{P}$ is reasonable because it is decidable over all finite inputs and $\hat{P}(\cdot)$. $\hat{P}$ is not a safety property because there exists an invalid execution (a_1) that can be extended to a valid execution $(a_1; r)$. $\hat{P}$ is also not a liveness property because there exists an invalid finite-length execution (a_2) that cannot be extended to a valid execution.

We next define an MRA M to enforce this reasonable nonsafety-nonliveness property. M operates simply by repeating the following: if the input action is a_1 then insert a_1 to be executed, obtain its result r, and output r to the target as the result of a_1; otherwise, if the input action is a_2 then enter an infinite loop (i.e., output no additional actions or results).

M is a sound enforcer with respect to $\hat{P}$ because its output executions will only ever be of the form $(a_1; r)^\infty$ (because it only outputs a_1 actions and r results, and it always outputs an r for every a_1 it outputs). M is also transparent because it never modifies valid input executions (because M outputs every a_1 and r it inputs). Hence, M correctly enforces this reasonable nonsafety-nonliveness property $\hat{P}$. □

There also exist some nonsafety, liveness properties enforceable by MRAs, as the following theorem shows.

Theorem 9. *There exists a reasonable property $\hat{P}$ on a system with event set E that is not a safety property but is a liveness property and can be enforced by an MRA M.*

Proof. Let $E = \{a, r\}$, where a is an action and r is a result. Also let $\hat{P}$ be satisfied by exactly those executions that match the pattern $(a; r)^\infty$. $\hat{P}$ is reasonable because it is decidable over all finite inputs and $\hat{P}(\cdot)$. $\hat{P}$ is not a safety property because there exists an invalid execution (a) that can be extended to a valid execution $(a; r)$. $\hat{P}$ is a liveness property because the only invalid finite-length executions have the form $(a; r)^*; a$, but any such execution can be extended into a valid execution by appending an r.

We next define an MRA M to enforce this reasonable nonsafety-liveness property. M operates simply by repeating the following: read the input action a, insert a, obtain its result r, and return r to the target. Thus, M executes every action the target attempts to execute, and M returns every result generated by the system to the target.

M is a sound enforcer with respect to $\hat{P}$ because for every a that it inputs, it outputs a, inputs r, and outputs r. Therefore, its output executions will only ever be of the form $(a; r)^\infty$. M is also transparent because it never modifies valid input executions (because M outputs every a and r it inputs). Hence, M correctly enforces this reasonable nonsafety-liveness property $\hat{P}$. □

There are some liveness properties unenforceable by MRAs, such as termination. An MRA cannot enforce termination because in order to be transparent, it must output all valid input executions, so for any infinite-length input execution (which must be invalid), the MRA would have to output all of its prefixes, implying that the MRA outputs the entire infinite-length input execution over its infinite-length run. Because no MRA can enforce this liveness property termination, there are also some nonsafety-nonliveness properties properties unenforceable by MRAs, which we can obtain by taking the conjunction of termination and any safety property; an MRA cannot not enforce such a nonsafety-nonliveness property

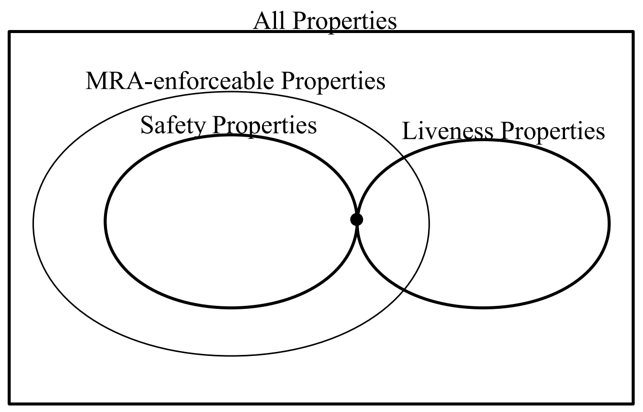

Figure 4.1. Relationships between safety, liveness, and MRA-enforceable properties.

for the same reason it cannot enforce the termination property alone. Figure 4.1 summarizes all of these findings.

CHAPTER 5

RELATED WORK AND CONCLUSIONS

5.1 Related Work

In the past decade several researchers have modeled runtime program monitors as security automata and analyzed their enforcement powers; a recent article surveys these results [3]. 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 [2]; Viswanathan, Kim, and others refined these safety bounds by adding computability constraints to the safety properties being enforced [16, 17]; Fong showed that truncation automata with limited memory (e.g., with space to store only a bounded number of actions seen) can still enforce practically useful safety policies [18]; Walker and Aktug et al. presented techniques for statically guaranteeing that policies represented by truncation automata get enforced at runtime [19, 20, 10]; 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 [21]; Le Guernic et al. and Hamlen et al. considered some enforcement capabilities of runtime security automata with access to source code [22, 23, 15]; Ligatti, Bauer, and Walker defined execution-transforming monitors called edit automata, defined policy enforcement in the presence of edit automata, and found that edit automata enforce reasonable renewal properties [3, 24]; 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 [25]) [26]. In addition, security automata have formed the basis of several policy-specification languages [7, 14, 8, 10].

Some of the security-automata-related research mentioned above has considered results of actions [20, 10, 21], 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 result-returns as actions poses no problems in these models because they consider monitors to be truncation automata, which always accept valid actions (and results) and halt the target when encountering an invalid action (or result). In contrast, this thesis gives monitors the practical ability to edit (i.e., transform) 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).

5.2 Conclusion

This thesis has presented a model of runtime monitors that improves on existing models by capturing the realistic constraint that monitors must return a result for one action before being able to see the next action (which the application may generate based on the result of the first action). Incorporating results into the model and obligating monitors to return (possibly `void` or `unit`) results for all monitored actions prevents the monitors from enforcing some properties, but monitors modeled by MRAs can nonetheless enforce a strict superset of the safety policies.

These findings are important because they improve our understanding of the sorts of policies we can ever hope to enforce dynamically in systems with synchronous actions. We hope that with continued research, we will be able to develop algorithms for decomposing general policies into statically and dynamically enforceable subpolicies, such that we could enforce a maximum set of policy constraints statically and the remaining constraints dynamically.

REFERENCES

[1] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

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

[3] Jay Ligatti, Lujo Bauer, and David Walker. Run-time enforcement of nonsafety policies. *ACM Transactions on Information and System Security*, 12(3):1–41, January 2009.

[4] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions of Software Engineering*, 3(2):125–143, 1977.

[5] Bowen Alpern and Fred Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1987.

[6] David Evans and Andrew Twyman. Flexible policy-directed code safety. In *IEEE Security and Privacy*, Oakland, CA, May 1999.

[7] Úlfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the New Security Paradigms Workshop*, pages 87–95, Caledon Hills, Canada, September 1999.

[8] Lujo Bauer, Jay Ligatti, and David Walker. Composing expressive runtime security policies. *ACM Transactions on Software Engineering and Methodology*, 18(3):1–43, 2009.

[9] Jay Ligatti, Billy Rickey, and Nalin 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.

[10] Irem Aktug and Katsiaryna Naliuka. ConSpec–a formal language for policy specification. In *Proceedings of the First International Workshop on Run Time Enforcement for Mobile and Distributed Systems*, September 2007.

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

[12] Jay Ligatti, Lujo Bauer, and David Walker. Enforcing non-safety security policies with program monitors. In *10th European Symposium on Research in Computer Security (ESORICS)*, Milan, Italy, September 2005.

[13] Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: Enforcement mechanisms for run-time security policies. Technical Report TR-681-03, Princeton University, May 2003.

[14] Úlfar Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Cornell University, November 2003.

[15] Kevin Hamlen, Greg Morrisett, and Fred B. Schneider. Computability classes for enforcement mechanisms. *ACM Transactions on Progamming Languages and Systems*, 28(1):175–205, January 2006.

[16] Mahesh Viswanathan. *Foundations for the Run-time Analysis of Software Systems.* PhD thesis, University of Pennsylvania, 2000.

[17] Moonjoo Kim, Sampath Kannan, Insup Lee, Oleg Sokolsky, and Mahesh Viswantathan. Computational analysis of run-time monitoring—fundamentals of Java-MaC. In *Run-time Verification*, June 2002.

[18] Philip W. L. Fong. Access control by tracking shallow execution history. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2004.

[19] David Walker. A type system for expressive security policies. In *Twenty-Seventh ACM Symposium on Principles of Programming Languages*, pages 254–267, Boston, January 2000.

[20] Irem Aktug, Mads Dam, and Dilian Gurov. Provably correct runtime monitoring. In *Proceedings of the 15th International Symposium on Formal Methods*, May 2008.

[21] Mads Dam, Bart Jacobs, Andreas Lundblad, and Frank Piessens. Security monitor inlining for multithreaded java. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, July 2009.

[22] Gurvan Le Guernic, Anindya Banerjee, Thomas Jensen, and David A. Schmidt. Automata-based confidentiality monitoring. In *Proceedings of the Asian Computing Science Conference (ASIAN)*, December 2006.

[23] Gurvan Le Guernic. Automaton-based confidentiality monitoring of concurrent programs. In *Proceedings of the Computer Security Foundations Symposium (CSF)*, pages 218–232, July 2007.

[24] Jay Ligatti. *Policy Enforcement via Program Monitoring.* PhD thesis, Princeton University, June 2006.

[25] Edward Chang, Zohar Manna, and Amir Pnueli. Characterization of temporal property classes. In *Proceedings of Automata, Languages and Programming, volume 623 of LNCS*, pages 474–486. Springer-Verlag, 1992.

[26] Yliès Falcone, Jean-Claude Fernandez, and Laurent 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.