Enforcing More with Less: Formalizing Target-Aware Run-Time Monitors

Yannis Mallios[1], Lujo Bauer[1], Dilsun Kaynar[1], and Jay Ligatti[2]

[1] Carnegie Mellon University, Pittsburgh, USA
{mallios,lbauer,dilsunk}@cmu.edu
[2] University of South Florida, Tampa, USA
ligatti@cse.usf.edu

Abstract. Run-time monitors ensure that untrusted software and system behavior adheres to a security policy. This paper defines an expressive formal framework, based on I/O automata, for modeling systems, policies, and run-time monitors in more detail than is typical. We explicitly model, for example, the environment, applications, and the interaction between them and monitors. The fidelity afforded by this framework allows us to explicitly formulate and study practical constraints on policy enforcement that were often only implicit in previous models, providing a more accurate view of what can be enforced by monitoring in practice. We introduce two definitions of enforcement, target-specific and generalized, that allow us to reason about practical monitoring scenarios. Finally, we provide some meta-theoretical comparison of these definitions and we apply them to investigate policy enforcement in scenarios where the monitor designer has knowledge of the target application and show how this can be exploited to make more efficient design choices.

1 Introduction

Today's computing climate is characterized by increasingly complex software systems and networks, and inventive and determined attackers. Hence, one of the major thrusts in the software industry and in computer security research has become to devise ways to *provably guarantee* that software does not behave in dangerous ways or, barring that, that such misbehavior is contained. Example guarantees could be that programs: only access memory that is allocated to them (memory safety); only jump to and execute valid code (control-flow integrity); and never send secret data over the network (a type of information flow).

A common mechanism for enforcing security policies on untrusted software is run-time monitoring. Run-time monitors observe the execution of untrusted applications or systems and ensure that their behavior adheres to a security policy. This type of enforcement mechanism is pervasive, and can be seen in operating systems, web browsers, firewalls, intrusion detection systems, etc. A common example of monitoring is system-call interposition (e.g., [12]): given an untrusted application and a set of security-relevant system calls, a monitor intercepts calls made by the application to the kernel, and enforces a security policy

A. Jøsang, P. Samarati, and M. Petrocchi (Eds.): STM 2012, LNCS 7783, pp. 17–32, 2013.

by taking remedial action when a call violates the policy (Fig. 1). In practice, several instantiations of monitors could be used in this scenario. Understanding and formally reasoning about each is as important as understanding the general concept, since it allows important details to be captured that might be lost at a higher level of abstraction. Two dimensions along which instantiations can differ are: (1) *the monitored interface*: monitors can mediate different parts of the communication between the application and the kernel, e.g., an input-sanitization monitor will mediate only inputs to the kernel (dashed lines in Fig. 1); and (2) *trace-modification capabilities*: monitors may have a variety of enforcement capabilities, from just terminating the application (e.g., when it tries to write to the password file), to being able to perform additional remedial actions (e.g., suppress a write system call and log the attempt).[1]

Despite the ubiquity of run-time monitors, their use has far outpaced theoretical work that makes it possible to formally and rigorously reason about monitors and the policies they enforce. Such theoretical work is necessary, however, if we are to have confidence that enforcement mechanisms are successfully carrying out their intended functions.

Several proposed formal models (e.g., [23,17]) make progress towards this goal. They use formal frameworks to model monitors and their enforcement capabilities, e.g., whether the monitors can insert arbitrary actions into the stream of actions that the target wants to execute. These frameworks have been used to analyze and characterize the policies that are enforceable by the various types of monitors.

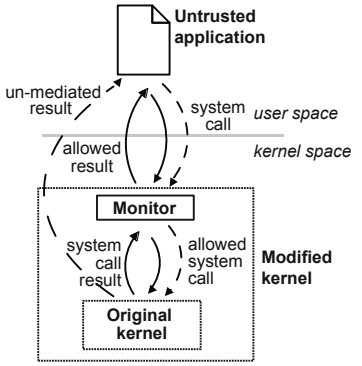


Fig. 1. System-call interposition: dashed line shows an input-mediating monitor; solid line an input/output-mediating monitor.

However, such models typically do not capture many details of the monitoring process, including the monitored interface, leaving us with practical scenarios that we cannot reason about in detail. In our system-call interposition scenario, for example, without the ability to model the communication between the untrusted application, the monitor, and the kernel, it may not be possible to distinguish between and compare monitors that can mediate all security-relevant communication between the application and the kernel (solid lines in Fig. 1) and monitors that can mediate only some of it (dashed lines in Fig. 1).

Some recent models (e.g., [18,13]) allow reasoning about bi-directional communication between the monitor and its environment (e.g., application and kernel), but do not explicitly reason about the application or system being monitored. In practice, however, monitors can enforce policies beyond their operational enforcement capabilities by exploiting knowledge about the component that they are monitoring. For example, a policy that requires that every file that is opened

[1] In this paper we do not consider mechanisms that arbitrarily modify the target application, such as by rewriting.

must be eventually closed cannot, in general, be enforced by any monitor, because the monitor cannot know what the untrusted application will do in the future. However, if the monitored application always closes files that it opens, then this policy is no longer unenforceable for that particular application. Such distinctions are often relevant in practice—e.g., when implementing a patch for a specific type or version of an application—and, thus, there is a need for formal frameworks that will aid in making informed and provably correct design and implementation decisions.

In this paper, we propose a general framework, based on I/O automata, for more detailed reasoning about policies, monitoring, and enforcement. The I/O automaton model [19] is a labeled transition model for asynchronous concurrent systems. We chose I/O automata because we wanted an automata-based formalism, similarly to many previous models of run-time enforcement mechanisms, but with enough expressive power and established meta-theory to model asynchronous systems (e.g., communication between the application, the monitor, and the kernel).Our framework provides abstractions for reasoning about many practical details important for run-time enforcement, and, in general, yields a richer view of monitors and applications than is typical in previous analyses of run-time monitoring. For example, our framework supports modeling practical systems with security-relevant actions that the monitor cannot mediate.

We make the following specific contributions:

- We show how I/O automata can be used to faithfully model target applications, monitors, and the environments in which monitored targets operate, as well as various monitoring architectures (§3).
- We extend previous definitions of security policies and enforcement to support more fine-grained formal reasoning of policy enforcement (§4).
- We show that this more detailed model of monitoring forces explicit reasoning about concerns that are important for designing run-time monitors in practice, but about which previous models often reasoned only informally (§5.2). We formalize these results as a set of lower bounds on the policies enforceable by any monitor in our framework.
- We demonstrate how to use our framework to exploit knowledge about the target application to make design and implementation choices that may lead to more efficient enforcement (§5.3). For example, we exhibit constraints under which monitors with different monitoring interfaces (i.e., one can mediate more actions than the other) can enforce the same class of policies.

Limitations. The goal of this paper is to introduce an expressive framework for reasoning about run-time monitors, and illustrate how to use it for meta-theoretical analysis of run-time enforcement. Due to space constraints, we omit or abbreviate several important aspects of the discussion, including: (1) Real-world examples: We provide simple examples to illustrate the use and advantages of our framework (§3), but do not encode any complex real-world scenarios. I/O automata-based modeling of complex applications requires mostly standalone work (e.g., [15,2]), and we defer to future work such examples for run-time monitoring. (2) Translation of previous models: We omit translations of

previous models of monitors (e.g., edit automata [17]) or auxiliary notions for defining enforcement (e.g., transparency [17]). Examples of modeling previously studied monitors can be found in our technical report [20]; a discussion of why *transparency* is non-essential in expressive frameworks, such as ours, can be found in [18]. (3) Formal presentation: When the notational complexity needed to formally define I/O automata and our framework obscures underlying insights, our presentation is not fully formal; a more formal treatment can be found in our technical report [20].

Roadmap. We first briefly review I/O automata (§2). We then informally show how to model monitors and targets in our framework and discuss some benefits of this approach (§3). Then, we formally define policies and enforcement (§4), and show examples of meta-theoretical analysis enabled by our framework: (a) lower bounds for enforceable policies (§5.2), and (b) constraints under which seemingly different monitoring architectures enforce the same classes of policies (§5.3).

2 I/O Automata

I/O automata are a labeled transition model for asynchronous concurrent systems [24,19]. Here we informally review aspects of I/O automata that we build on in the rest of the paper; please see our technical report [20] for a more formal treatment. We encourage readers familiar with I/O automata to skip to §3.

I/O automata are typically used to describe the behavior of a system interacting with its environment. The interface between an automaton A and its environment is described by the ***action signature*** $sig(A)$, a triple of disjoint sets—$input(A)$, $output(A)$, and $internal(A)$. We write $acts(A)$ for $input(A) \cup output(A) \cup internal(A)$, and call output and internal actions *locally controlled*.

Formally, an I/O automaton A consists of: (1) an action signature, $sig(A)$; (2) a (possibly infinite) set of *states*, $states(A)$; (3) a nonempty set of *start states*, $start(A) \subseteq states(A)$; (4) a transition relation, $trans(A) \subseteq states(A) \times acts(A) \times states(A)$, such that for every state q and input action a there is a transition $(q, a, q') \in trans(A)$; and (5) an equivalence relation $Tasks(A)$ partitioning the set $output(A) \cup internal(A)$ into at most countably many equivalence classes.

If A has a transition (q, a, q') then we say that action a is *enabled* in state q. Since every input action is enabled in every state, I/O automata are said to be *input-enabled*. When only input actions are enabled in q, then q is called a *quiescent* state. The set of all quiescent states of an automaton A is denoted by $quiescent(A)$. The equivalence relation $Tasks(A)$ is used to define *fairness*, which essentially says that the automaton will give fair turns to each of its tasks while executing.

An ***execution*** e of A is a finite sequence, $q_0, a_1, q_1, \ldots, a_r, q_r$, or an infinite sequence $q_0, a_1, q_1, \ldots, a_r, q_r, \ldots$, of alternating states and actions such that $(q_k, a_{k+1}, q_{k+1}) \in trans(A)$ for $k \geq 0$, and $q_0 \in start(A)$. A ***schedule*** is an execution without states in the sequence, and a ***trace*** is a schedule that consists only of input and output actions. An *execution*, *trace*, or *schedule* ***module*** describes the behavior exhibited by an automaton. An execution module E consists of

a set $states(E)$, an action signature $sig(E)$, and a set $execs(E)$ of executions. Schedule and trace modules are similar, but do not include states. The sets of executions, schedules, and traces of an I/O automaton (or module) X are denoted by $execs(X)$, $scheds(X)$, and $traces(X)$. Given a sequence s and a set X, $s|X$ denotes the sequence resulting from removing from s all elements that do not belong in X. Similarly, for a set of sequences S, $S|X = \{(s|X) \mid s \in S\}$.

The symbol ϵ denotes the empty sequence. We write $\sigma_1;\sigma_2$ for the concatenation of two schedules or traces, the first of which has finite length. When σ_1 is a finite prefix of σ_2, we write $\sigma_1 \preceq \sigma_2$, and, if a strict finite prefix, $\sigma_1 \prec \sigma_2$. $\Sigma^\star$ denotes the set of finite sequences of actions and Σ^ω the set of infinite sequences of actions. The set of all finite and infinite sequences of actions is $\Sigma^\infty = \Sigma^\star \cup \Sigma^\omega$.

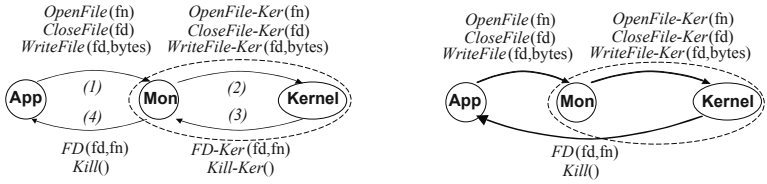
An automaton that models a complex system can be constructed by *composing* automata that model the system's components. A set of I/O automata is *compatible* if their output actions are disjoint and the internal actions of each automaton are disjoint with all actions of the other automata. The composition $A = A_1 \times \ldots \times A_n$ of a set of compatible automata $\{A_i : i \in I\}$ is the automaton that has as states the Cartesian product of the states of the component automata and as behaviors the interleavings of the behaviors of the component automata. Composition of modules is defined similarly [24].

Unlike in models such as CCS [22], composing two automata that share actions (i.e., outputs of one are inputs to the other) causes those actions to become output actions of the composition. Actions that are required to be internal need to be explicitly classified as such using the *hiding* operation, which takes as input a signature and a set of output actions to be hidden, and produces a signature with those actions reclassfied as internal. *Renaming*, on the other hand, changes the names of actions, but not their types, i.e., renaming is a total injective mapping between sets of actions.

3 Specifying Targets and Monitors

We model targets (the entities to be monitored) and monitors as I/O automata, and denote them by metavariables $\mathcal{T}$ and $\mathcal{M}$. Targets composed with monitors are *monitored targets*; such an example is the modified kernel in Fig. 1. Monitored targets may themselves be targets for other monitors.

Building on the system-call-interposition example (Fig. 1), we now show how to model monitors and targets using I/O automata. Suppose the application's only actions are *OpenFile*, *WriteFile*, and *CloseFile* system calls; the kernel's actions are *FD* (to return a file descriptor) and the *Kill* system call. The application can request to open a file *fn*, and the kernel keeps track of the requests as part of its state. When a file descriptor *fd* is returned in response to a request for *fn*, *fn* is removed from the set of requests. The application can then write *bytes* number of bytes to, or close, *fd*. Finally, a *Kill* action terminates the application and clears all requests. Such a formalization, where the target's actions depend on results returned by the environment, was outside the scope of original run-time-monitor models, as identified also by recent work (e.g., [18,13]).



(a) Input/output-mediating monitor (b) Input-mediating monitor

Fig. 2. I/O automata interface diagrams of kernel, application, and monitor

Fig. 2a shows I/O automata interface diagrams of the monitored system consisting of the application and the monitored kernel.[2] The application's and the kernel's interfaces differ only in that the input actions of the kernel are output actions of the application, and vice versa. This models the communication between the application and the kernel when they are considered as a single system. The kernel's readiness to always accept file-open requests is modeled naturally by the input-enabledness of the I/O automaton. Paths (2) and (3) represent communication between the monitor and the kernel through the *renamed* actions of the kernel (using the renaming operation of I/O automata), e.g., *OpenFile*(x) becomes *OpenFile-Ker*(x), and thus irrelevant to a policy that reasons about *OpenFile* actions. Renaming models changing the target's interface to allow the monitor to intercept the target's actions. In practice, this is often done by rewriting the target to inline a monitor. Finally, we also *hide* the communication between the monitor and the kernel to keep it internal to the monitored target (denoted by the dotted line around the monitored kernel automaton). This models a monitoring process that is transparent to the application (i.e., the application remains unaware that the kernel is monitored).

In our system-call interposition example we described some choices that a monitor designer can make, such as choosing (1) the interface to be monitored, e.g., mediate only input actions, and (2) the trace-modification capabilities of the monitor. Choice (1) can be expressed in our model by appropriately restricting the renaming function applied to the target. For example, in Fig. 2b, we renamed only the input actions of the kernel (i.e., *OpenFile*, *CloseFile*, and *WriteFile*). This models monitors that mediate inputs sent to the target and can prevent, for example, SQL injections attacks. Similarly, renaming only the outputs of the target models monitors that mediate only output actions (and can prevent, for example, cross-site scripting attacks). Choice (2) is closely related to representing previous models of monitors in our framework; please see our technical report for more detail on this and on modeling different trace-modification capabilities [20].

4 Policy Enforcement

In this section we define security policies and two notions of enforcement: target-specific and generalized enforcement.

[2] The kernel's I/O automaton definition is shown in our technical report [20].

4.1 Security Policies

A *policy* is a set of (execution, schedule, or trace[3]) modules. We let the metavariables $\mathscr{P}$ and $\hat{P}$ range over policies and their elements, i.e., modules, respectively. The novelty of this definition of policy compared to previous ones (e.g., [23,17]) is that each element of the policy is not a set of automaton runs, but, rather, a pair of a set of runs (i.e., schedules or traces) and a signature, which is a triple consisting of a set of inputs, a set of outputs, and a set of internal actions. The signature describes explicitly which actions that do not appear in the set of runs are relevant to a policy. This is useful in a number of ways. When enforcing a policy on a system composed of several previously defined components, for example, the signatures can clarify whether a policy that is being enforced on one component also reasons about (e.g., prohibits or simply does not care about) the actions of another component. For our running example, if the signature contains only *Open*, *FD*, and *Kill*, then all other system calls are security irrelevant and thus permitted; if the signature contains other system calls (*SocketRead*), then any behaviors exhibiting those calls are prohibited.

Our definition of a policy as a set of modules resembles that of a hyperproperty [8] and previous definition of policies (modulo the signature of each schedule or trace module) and captures common types of policies such as access control, noninterference, information flow, and availability.

I/O automata can have infinite states and produce possibly infinite computations. We would like to avoid discussions of computability and complexity issues, and so we make the assumption that all policies $\mathscr{P}$ that we discuss are *implementable* [24], meaning that for each module $\hat{P}$ in $\mathscr{P}$, there exists an I/O automaton A such that $sig(A) = sig(\hat{P})$ and $scheds(\hat{P}) \subseteq scheds(A)$.

4.2 Enforcement

In §3 we showed how monitoring can be modeled by renaming a target $\mathcal{T}$ so that its security-relevant actions can be observed by a monitor $\mathcal{M}$ and by hiding actions that represent communication unobservable outside of the monitored target. We now define enforcement formally as a relation between the behaviors allowed by the policy and the behaviors exhibited by the monitored target.

Definition 1 *(Target-specific enforcement) Given a policy $\mathscr{P}$, a target $\mathcal{T}$, and a monitor $\mathcal{M}$ we say that $\mathscr{P}$ is* ***specifically soundly enforceable*** *on $\mathcal{T}$ by $\mathcal{M}$ if and only if there exists a module $\hat{P} \in \mathscr{P}$, a renaming function* rename*, and a hiding function* hide *for some set of actions Φ such that $(scheds(\mathsf{hide}_\Phi(\mathcal{M} \times \mathsf{rename}(\mathcal{T}))) | acts(\hat{P})) \subseteq scheds(\hat{P})$.*

Here, $\mathsf{hide}_\Phi(\mathcal{M} \times \mathsf{rename}(\mathcal{T}))$ is the monitored target: the target $\mathcal{T}$ is renamed so that its security-relevant actions can be observed by the monitor $\mathcal{M}$; *hide* is applied to their composition to prevent communication between the monitor

[3] Our analyses equally apply to execution modules, but, for brevity, we discuss only schedule and trace modules.

and the target from leaking outside the composition.[4] If a target does not need renaming, rename can be the identity function; if we do not care about hiding all communication, *hide* can apply to only some actions. For example, suppose the monitored target from our running example (node with dotted lines in Fig. 2b) is composed with an additional monitor that logs system-call requests and responses. We would then keep the actions for system-call requests and responses visible to the logging monitor by not hiding them in the initial monitored target.

Def. 1 binds the enforcement of a policy by a monitor to a specific target. We refer this type of enforcement as *target-specific enforcement* and to the corresponding monitor as a target-specific monitor. However, some monitors may be able to enforce a property on any target. One such example is a system-call-interposition mechanism that operates independently of the target kernel's version or type (e.g., a single monitor binary that can be installed in both Windows and Linux). We call this type of enforcement *generalized enforcement*, and the corresponding monitor a generalized monitor.[5] More formally:

Definition 2 *(Generalized enforcement) Given a policy* $\mathscr{P}$ *and a monitor* $\mathcal{M}$ *we say that* $\mathscr{P}$ *is* ***generally soundly enforceable*** *by* $\mathcal{M}$ *if and only if for all targets* $\mathcal{T}$ *there exists a module* $\hat{P} \in \mathscr{P}$*, a renaming function* rename*, such that* $(scheds(\mathcal{M} \times \mathsf{rename}(\mathcal{T}))|acts(\hat{P})) \subseteq scheds(\hat{P})$*.*

Different versions of Def. 1 and 2 can be obtained by replacing schedules with traces (trace enforcement), fair schedules, or fair traces (fair enforcement); or by replacing the subset relation with other set relations (e.g., equality) for comparing the behaviors of the monitored target to those of the policy [18,5]. In this paper we focus on the subset and equality relations, and refer to the corresponding enforcement notions as *sound* (e.g., Def. 1) and *precise enforcement*.

No constraints are placed on the renaming and hiding functions in Def. 1 and 2, as this permits more enforcement scenarios to be encoded in our framework. For example, by α-renaming a target, which in practice means that we are ignoring it, and incorporating some of its functionality (if needed) in a monitor, we can encode a technique similar to the one used to automate monitor synthesis (e.g., [21]). Another reason for this choice, stemming from a meta-theoretical analysis of the two distinct notions of enforcement, is discussed next, in §4.3.

4.3 Comparing Enforcement Definitions

As an example of meta-theoretic analysis in our framework, we compare Def. 1 and 2. More specifically, one might expect target-specific monitors to have an advantage in enforcement, i.e., the class of target-specific monitors *should* enforce

[4] Since Def. 1 reasons about schedules (i.e., internal actions as well as input and output), hide_Φ is redundant: hiding reclassifies some output actions as internal, but does not remove them from schedules. We include it here to illustrate our framework's ability to expose that, in practice, the environment can be oblivious to the monitoring of the target application. In the rest of the paper we will omit the hiding operator.

[5] Monitors of previous models, such as [23] and [17], are generalized monitors.

a larger set of policies than the class of generalized monitors. Intuitively, this is because if a policy describes a behavior that is correctable for specific targets, but not for all targets, then there is no generalized monitor that can enforce that policy, even if target-specific monitors exist.[6]

Proposition 1. *Given a monitor* $\mathcal{M}$*:*

1. $\forall \mathscr{P} : \mathscr{P}$ *is* ***generally soundly enforceable*** *by* $\mathcal{M} \Rightarrow$
   $\forall \mathcal{T} : \mathscr{P}$ *is* ***specifically soundly enforceable*** *on* $\mathcal{T}$ *by* $\mathcal{M}$*, and*
2. $\exists \mathscr{P} \exists \mathcal{T} : (\mathscr{P}$ *is* ***specifically soundly enforceable*** *on* $\mathcal{T}$ *by* $\mathcal{M}) \wedge$
   $\neg(\mathscr{P}$ *is* ***generally soundly enforceable*** *by* $\mathcal{M})$.

Prop. 1 compares the definitions of enforcement (Def. 1 and 2) with respect to the same monitor, and shows that they capture the intuition that a monitor that enforces a policy without being tailored for a specific target can enforce the policy on any target, while the inverse does not hold in general.

However, we can get a deeper insight when trying to characterize the two definitions of enforcement in general, i.e., independently of a specific monitor. Surprisingly, in such a comparison the two definitions turn out to be equivalent.

Theorem 1. $\forall \mathscr{P} \forall \mathcal{T}$*:*
$\exists \mathcal{M} : \mathscr{P}$ *is* ***specifically soundly enforceable*** *on* $\mathcal{T}$ *by* $\mathcal{M} \Leftrightarrow$
$\exists \mathcal{M}' : \mathscr{P}$ *is* ***generally soundly enforceable*** *by* $\mathcal{M}'$.

The right-to-left direction of the theorem is straightforward: any generalized monitor can be used as a target-specific monitor. The other direction is more interesting since it suggests, perhaps surprisingly, that a generalized monitor can be constructed from a target-specific one. More specifically, given a monitor that enforces a policy on a specific target, we can use this *monitored target* as the basis for a monitor for any other target. In that case, security-relevant behaviors of the system would be exhibited only by the monitor (formally, all target actions would be renamed to become security irrelevant). For example, suppose that different versions of an application are installed on each of our machines. If we find a patch (i.e., monitor) for one version, then Thm. 1 implies that instead of finding patches for all other versions, we can distribute the patched version (i.e., monitored target) to all machines and modify the existing applications on those machines so that their behavior is ignored. This approach may apply if installing the patched version of the application on top of other versions is more cost-efficient than finding patches for all other versions.

Thm. 1 holds because Def. 1 and 2 place no restrictions on renaming functions (i.e., on how a monitor is integrated with a target). In practice, this interaction may be constrained. Thus, one might argue that it would be more natural for the only-if direction of the theorem to fail, since it erases the distinction between target-specific and generalized enforcement. The following theorem shows the effect of introducing such a constraint.

[6] Due to space constraints, proofs are given in our companion technical report [20].

Theorem 2. $\exists\mathscr{P}\exists\mathcal{T}$:
$\Big(\exists\mathcal{M}$: $\mathscr{P}$ *is* ***specifically soundly enforceable*** *on* $\mathcal{T}$ *by* $\mathcal{M}\Big)$ ***and***
$\neg\Big(\exists\mathcal{M}'$: $\mathscr{P}$ *is conditionally* ***generally soundly enforceable*** *by* $\mathcal{M}'$*, i.e., for all targets* $\mathcal{T}'$ *there exists a module* $\hat{P} \in \mathscr{P}$*, and a renaming function* rename*, such that if :*
$(\boldsymbol{C1})$ $acts(\hat{P}) \cap acts(\mathcal{T}') \neq \emptyset$*, and*
$(\boldsymbol{C2})$ $range(\mathsf{rename}) \subseteq acts(\mathcal{M}')$
then $(scheds(\mathcal{M}' \times \mathsf{rename}(\mathcal{T}'))|acts(\hat{P})) \subseteq scheds(\hat{P}))\Big)$.

The first condition (C1) prohibits using an element of the policy that is irrelevant to the target that we are trying to monitor. For example, if a policy consists of one module that contains networking events (i.e., the signature of the module contains only network-related actions) and another that contains file-related events (e.g., a signature similar to the one of our system-call interposition example), then if we want to enforce that policy on a network card (i.e., a target that exhibits network actions but no file actions), we must use the former module.

The second condition (C2) ensures that the only way that we rename the target is to match the monitors interface. In other words, we may not arbitrarily rename the target so that nobody can "listen" to its actions.

Thm. 2 lists some conditions under which generalized and target-specific enforcement are not equivalent. More satisfying would be to identify a single constraint under which the equivalence would not hold for any policy or target; however, given the wide variety of enforcement scenarios in practice, we conjecture that no single constraint allows Thm. 2 to be universally quantified over all policies and targets, as Thm. 1 is. Since our goal is to introduce a framework general enough to accommodate as many practical scenarios as possible, we rely on the monitor designer to impose appropriate restrictions on renaming or monitors that accurately reflect the specific practical scenarios under scrutiny.

5 Bounds on Enforceable Policies

This section describes several meta-theoretic results, facilitated by the abstractions described thus far, that further our understanding of the general limitations of practical monitors that fit this model.

5.1 Auxiliary Definitions

I/O automata are input enabled—all input actions are enabled at all states. Several arguments can be made in favor of or against input-enabledness. For example, one can argue that input-enabledness leads to better design of systems because one has to consider all possible inputs [19]. On the other hand, this constraint may be too restrictive for practical systems [1].

In our context, we believe input-enabledness is a useful characteristic, since run-time monitors are by nature input-enabled systems: a monitor may receive input at any time both from the target and from the environment (e.g., keyboard or network). However, a monitor modeled as an input-enabled automaton can enforce only those policies that allow the arrival of inputs at any point during execution. This is reasonable: a policy that prohibits certain inputs cannot be enforced by a monitor that cannot control those inputs. We later combine this and other constraints to describe the lower bound of enforceability in our setting.

We say that a module (or policy) is *input forgiving* (respectively, *internal* and *output* forgiving) if and only if it allows the empty sequence and allows each valid sequence to be extended to another valid sequence by appending any (possibly infinite) sequence of inputs.

Definition 3 *A schedule module* $\hat{P}$ *is* ***input forgiving*** *if and only if:*
(1) $\epsilon \in scheds(\hat{P})$*; and*
(2) $\forall s_1 \in scheds(\hat{P}) : \forall s_2 \preceq s_1 : \forall s_3 \in (input(\hat{P}))^\infty : (s_2; s_3) \in scheds(\hat{P})$.

I/O automata's definition of executions allows computation to stop at any point. Thus, the behavior of an I/O automaton is *prefix-closed*: any prefix of an execution exhibited by an automaton is also an execution of that automaton.

Definition 4 *A schedule module* $\hat{P}$ *is* ***prefix closed*** *if and only if:*
$\forall s_1 \in \Sigma^\infty : \Big(s_1 \in scheds(\hat{P}) \Rightarrow \forall s_2 \in \Sigma^\star : s_2 \preceq s_1 : s_2 \in scheds(\hat{P})\Big)$.

These two characteristics are unsurprising from the standpoint of models for distributed computation, but describe practically relevant details typically absent from models of run-time enforcement. Our model, instead of making assumptions that may not hold in practice, e.g., that all actions can be mediated, takes a more nuanced view, which admits that some aspects of enforcement are outside the monitor's control, such as scheduling strategies for security-relevant actions that cannot be mediated. The above definitions help explicate these assumptions when reasoning about enforceable policies, as we see next.

5.2 Lower Bounds of Enforceable Policies

Another constraint that affects enforceability and is specific to monitoring is that, in practice, a monitor cannot always ignore *all* behavior of the target application. Some real monitors decide what input actions the application sees, but otherwise do not interfere with the application's behavior—firewalls belong to this class of monitors. In such cases, a monitor can soundly enforce a policy only if the policy allows all behaviors that the target can exhibit when it receives no input. We call such policies, and their modules, *quiescent forgiving* (recall the definition of a quiescent state from §2). This captures a limitation that was understood to be present in run-time monitoring, but that typically was not formally expressed. More formally:

Definition 5 *A schedule module* $\hat{P}$ *is* **quiescent forgiving** *for some* $\mathcal{T}$ *if and only if:*
$\forall e \in execs(\mathcal{T})$ *such that* $e = q_0, a_1, \ldots, q_n$:
$\Big(q_n \in quiescent(\mathcal{T}) \wedge \big(\forall i \in \mathbb{N} : 0 \leq i < n : q_i \notin quiescent(\mathcal{T})\big)\Big) \Rightarrow$
$\big(sched(e)|acts(\hat{P})\big) \in scheds(\hat{P}) \wedge (\forall i \in \mathbb{N} : 0 \leq i < n : (sched(q_0, \ldots, q_i) \mid acts(\hat{P})) \in scheds(\hat{P}))$.

The following theorem formalizes a lower bound: a policy that is not quiescent forgiving, input forgiving, and prefix closed cannot be (precisely) enforced.

Theorem 3. $\forall \mathscr{P} : \forall \hat{P} \in \mathscr{P} : \forall \mathcal{T} : \forall \mathsf{rename} :$
$\exists \mathcal{M} : (scheds(\mathcal{M} \times \mathsf{rename}(\mathcal{T}))|acts(\hat{P})) = scheds(\hat{P}) \Rightarrow$
$\hat{P}$ *is input forgiving, prefix closed, and quiescent forgiving for* $\mathsf{rename}(\mathcal{T})$.

Thm. 3 reveals that monitors, regardless of their editing power, can enforce only prefix-closed properties (e.g., safety). In our context, even the equivalent of an edit monitor cannot enforce renewal properties (unlike in [17]), since, when constrained by prefix closure, renewal properties collapse to safety. This is because our model of executions allows computation to stop at any point, highlighting that the system may stop executing for reasons beyond our control, e.g., a power outage. In contrast, previous models assumed that a monitor's enabled actions would always be performed (e.g., [17]). In our framework, such guarantees are not built in, but can be explicitly added through fairness and other similar constraints on I/O automata [16]. This is another instance of our framework making explicit practical assumptions and constraints that affect enforcement. Earlier results about the enforcement powers of different types of monitors (e.g., that truncation monitors enforce safety and edit monitors enforce renewal policies) are also provable in our framework when we restrict reasoning to fair schedules and traces.

In practice, monitors typically reproduce at most a subset of a target's functionality. If a monitor composed with an application is to exhibit the same range of behaviors as the unmonitored application, it typically consults the target to generate these behaviors. In the system-call interposition example, for instance, the monitor cannot return correct file descriptors without consulting the kernel. Such monitors, which regularly consult an application, cannot precisely enforce (with respect to schedules) arbitrary policies even if they are quiescent forgiving, input forgiving, and prefix-closed. This is because input the monitor forwards to an application may cause the application to execute internal or output actions (e.g., a buffer overflow) forbiden by the policy, but which the monitor cannot prevent, since these are outside the interface between the monitor and the target.

On the other hand, in practice it is also common for the monitor (or system designer) to have some knowledge about the target. This knowledge can be exploited to use simpler-than-expected monitors to enforce (seemingly) complex policies. Although similar observations have been made before (e.g., program re-writing [14], non-uniformity [17], use of static analysis [7]), our approach formalizes them within a single framework, which allows new results that were beyond the scope of previous work, as we demonstrate in the following section.

5.3 Policies Enforceable by Target-Specific Monitors

As discussed in §3, our framework allows defining monitors that mediate different parts of the communication of a target with its environment, e.g., mediating a target's inputs and outputs or just its outputs. For brevity, rather than analyzing the policies enforceable by specific monitors, as done previously [23,17,18,14], we show an instance in which our framework enables formal results useful to designers of run-time monitors who have knowledge about the target application. This is a novel analysis of how some knowledge of the target can compensate (in terms of enforceability) for a narrower monitoring interface.[7]

In §3 we described two monitoring architectures: one where the monitor mediates the inputs and the outputs of the target, and another where it mediates just the inputs. Intuitively, an input/output-mediating monitor should be able to enforce a larger class of policies than an input-mediating one, since the former is able to control (potentially) more security-relevant actions than the latter (i.e., the outputs of the target). In other words, some policies should be enforceable by input/output mediating monitors, but not by input mediating ones:[8]

Theorem 4. $\exists \mathscr{P}$:
($\mathscr{P}$ *is generally precisely enforceable by some input/output-mediating* $\mathcal{M}_1$)$\wedge$
$\neg$($\mathscr{P}$ *is generally precisely enforceable by some input-mediating* $\mathcal{M}_2$).

For the proof we pick a policy whose elements (i.e., modules) disallow all output actions (excluding the ones used for target-monitor communication) and a target that performs only output actions. An input-mediating monitor cannot enforce that policy on that target since it does not mediate its output, and thus it cannot generally enforce the policy. However, an input/output-mediating monitor will be able to enforce the policy, since whenever it receives any (renamed) actions from the target, it will just suppress them.

Thm. 4 establishes that some policies are generally enforceable by input/output-mediating monitors but not by input-mediating monitors. However, for some targets the two architectures are equivalent in enforcement power:

Theorem 5. $\forall \mathscr{P} : \forall \mathcal{T}$:
$\mathscr{P}$ *is specifically precisely enforceable on* $\mathcal{T}$ *by some input/output-mediating* $\mathcal{M}_1$
iff $\mathscr{P}$ *is specifically precisely enforceable on* $\mathcal{T}$ *by some input-mediating* $\mathcal{M}_2$
given that:

- ***(C1)*** $\mathscr{P}$ *does not reason about the communication between the monitor and the target,*
- ***(C2)*** $\forall \hat{P} \in \mathscr{P} : scheds(\hat{P})$ *is quiescent forgiving for* $\mathcal{T}$*,*
- ***(C3)*** $\forall \hat{P} \in \mathscr{P} : scheds(\hat{P}) \subseteq scheds(\mathcal{T})$*, i.e., the policy does not allow schedules that cannot be exhibited by the target, and*

[7] Another instance that focuses on the trace-modification capabilities of monitors and shows how arguments that were difficult to formalize in less expressive frameworks (e.g., [17]) can be naturally discussed in our model, can be found in [20].

[8] For brevity, we state theorems informally, omitting details necessary for the proofs. See our technical report [20] for detailed versions of the theorems and the proofs.

$$(\boldsymbol{C4})\ \forall \hat{P} \in \mathscr{P}\colon \forall s \in scheds(\mathcal{T}) : \Big(s \notin scheds(\hat{P}) \Rightarrow \exists s' \preceq s : \Big((s' \in scheds(\hat{P})) \\ \wedge\ (s' = s''; a) \wedge (a \in input(\mathcal{T})) \\ \wedge\ (\forall t \succeq s'\colon t \in scheds(\mathcal{T}) \Rightarrow t \notin scheds(\hat{P}))\Big)\Big).$$

Constraint C2 ensures that a target's outputs at the beginning of its execution, until it blocks for input, obey the policy. C3 ensures that an input/output-mediating monitor does not have an "unfair" advantage over an input-mediating one due to policies that require actions that a target cannot exhibit. C4 ensures that if a target receives input (from the monitor), then no behavior that it exhibits (until it blocks to wait for more input) will violate the policy; or, if it does, then that behavior can be suppressed without affecting the target's future behavior.

Thm. 5 shows how our framework can help to make sound decisions for designing monitors in practice. For example, consider a Unix kernel and the policy that a secret file cannot be (a) deleted or (b) displayed to guest users. To *precisely* enforce that policy, a monitor designer cannot in general use an input-mediating monitor: although it can enforce (a) by not forwarding commands like "rm secret-file", it cannot enforce (b), since it does not know whether the kernel can, for example, correctly identify guest users. However, if the designer knows that the specific kernel meets the constraints of Thm. 5, e.g., the kernel does not display secret files while booting (i.e., **C2**) and implements correct access-control for guest users (i.e., **C4**), then an input-mediating monitor suffices. The correctness of such design choices is not always obvious, and the above example shows how our framework can aid in making more informed decisions. Moreover, such decisions can benefit both efficiency (by not monitoring the kernel's output at run time) and security (since the monitor's TCB is smaller).

6 Related Work

The first model of run-time monitors, *security automata*, was based on Büchi Automata [23]. Since then, several similar models have extended or refined the class of enforceable policies based on the enforcement and computational powers of monitors (e.g., [17,11,10,4]). Unlike these works, we focus on modeling not just monitors, but also the target and the environment that monitors communicate with; this allows us to extend previous analyses of enforceable policies in ways that were out of the scope for previous frameworks [18].

Hamlen et al. described a model based on Turing Machines [14], and compared the policies enforceable by several types of enforcement mechanisms, including static analysis and inlined monitors. The main differences between this model and ours is that we explicitly model communication between the monitor, the target, and the environment, and we do not consider rewriting the target.

Recent work has revised these models or adopted alternate ones, such as the Calculus of Communicating Systems (CCS) [22] and Communicating Sequential Processes (CSP) [6], to more conveniently reason about applications, the interaction between applications and monitors, and enforcement in distributed systems. An example of a revised model are Mandatory Results Automata (MRA),

which model the (synchronous) communication between the monitor and the target [18]. MRA's, however, do not model the target explicitly, making it difficult to derive results about enforceable policies in target-specific environments.

Among the works building on CCS or CSP is Martinelli and Matteucci's model of run-time monitors based on CCS [21]. Like ours, their model captures the communication between the monitor and the target, but their main focus is on synthesizing run-time monitors from policies. In contrast, we focus on a meta-theoretical analysis of enforcement in a more expressive framework.

Basin et al. proposed a practical language, based on CSP and Object-Z (OZ), for specifying security automata [3]. They focus on the synchronization between a single monitor and target application, although the language can capture many other enforcement scenarios. Our work is similar, but focuses on showing how to use such an expressive framework to derive meta-theoretical results on enforceable policies in different scenarios, instead of on the (complementary aspect) of showing how to faithfully translate and model practical scenarios.

Gay et al. introduced *service automata*, a framework based on CSP for enforcing security requirements in distributed systems [13]. Although CSP provides the abstractions for reasoning about specific targets and communication with the monitor, such investigation and analysis is not the focus of that work.

7 Conclusion

Formal models of run-time monitors have improved our understanding of the powers and limitations of enforcement mechanisms [23,17], and aided in their design and implementation [9]. However, these models often fail to capture details relevant to real-world monitors, such as how monitors integrate with targets, and the extent to which monitors can control targets and their environment.

In this paper, we propose a general framework, based on I/O automata, for reasoning about policies, monitoring, and enforcement. This framework provides abstractions for reasoning about many practically relevant details important for run-time enforcement, and, in general, yields a richer view of monitors and applications than is typical in previous analyses. We also show how this framework can be used for meta-theoretic analysis of enforceable security policies. For example, we derive lower bounds on enforceable policies that are independent of the choice of monitor (Thm. 3). We also identify constraints under which monitors with different monitoring capabilities (e.g., monitors that see only a subset of the target's actions) can enforce the same classes of policies (Thm. 5).

Acknowledgements. This work was supported in part by NSF grants CNS-0716343, CNS-0742736, and CCF-0917047; and by Carnegie Mellon CyLab under Army Research Office grant DAAD19-02-1-0389.

References

1. de Alfaro, L., Henzinger, T.A.: Interface automata. In: Proc. European Software Engineering Conference (2001)

2. Araragi, T., Attie, P.C., Keidar, I., Kogure, K., Luchangco, V., Lynch, N.A., Mano, K.: On Formal Modeling of Agent Computations. In: Rash, J.L., et al. (eds.) FAABS 2000. LNCS (LNAI), vol. 1871, pp. 48–62. Springer, Heidelberg (2001)
3. Basin, D., Olderog, E.R., Sevinc, P.E.: Specifying and analyzing security automata using CSP-OZ. In: Proc. ACM Symposium on Information, Computer and Communications Security, ASIACCS (2007)
4. Basin, D., Jugé, V., Klaedtke, F., Zălinescu, E.: Enforceable Security Policies Revisited. In: Degano, P., Guttman, J.D. (eds.) Principles of Security and Trust. LNCS, vol. 7215, pp. 309–328. Springer, Heidelberg (2012)
5. Bishop, M.: Computer Security: Art and Science. Addison-Wesley Professional (2002)
6. Brookes, S.D., Hoare, C.A.R., Roscoe, A.W.: A theory of communicating sequential processes. Journal of the ACM 31, 560–599 (1984)
7. Chabot, H., Khoury, R., Tawbi, N.: Extending the enforcement power of truncation monitors using static analysis. Computers and Security 30(4), 194–207 (2011)
8. Clarkson, M.R., Schneider, F.B.: Hyperproperties. In: Proc. IEEE Computer Security Foundations Symposium (2008)
9. Erlingsson, U., Schneider, F.B.: SASI enforcement of security policies: a retrospective. In: Proc. Workshop on New Security Paradigms (2000)
10. Falcone, Y., Fernandez, J.C., Mounier, L.: What can you verify and enforce at runtime? Intl. Jrnl. Software Tools for Tech. Transfer (STTT) 14(3), 349–382 (2011)
11. Fong, P.W.L.: Access control by tracking shallow execution history. In: Proc. IEEE Symposium on Security and Privacy (2004)
12. Garfinkel, T.: Traps and pitfalls: Practical problems in system call interposition based security tools. In: Proc. Network and Distributed Systems Security Symposium (2003)
13. Gay, R., Mantel, H., Sprick, B.: Service Automata. In: Barthe, G., Datta, A., Etalle, S. (eds.) FAST 2011. LNCS, vol. 7140, pp. 148–163. Springer, Heidelberg (2012)
14. Hamlen, K.W., Morrisett, G., Schneider, F.B.: Computability classes for enforcement mechanisms. ACM Trans. Program. Lang. Syst. 28(1), 175–205 (2006)
15. Hickey, J., Lynch, N.A., van Renesse, R.: Specifications and Proofs for Ensemble Layers. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 119–134. Springer, Heidelberg (1999)
16. Kwiatkowska, M.Z.: Survey of fairness notions. Information and Software Technology 31(7), 371–386 (1989)
17. Ligatti, J., Bauer, L., Walker, D.: Run-time enforcement of nonsafety policies. ACM Transactions on Information and System Security 12(3) (2009)
18. Ligatti, J., Reddy, S.: A Theory of Runtime Enforcement, with Results. In: Gritzalis, D., Preneel, B., Theoharidou, M. (eds.) ESORICS 2010. LNCS, vol. 6345, pp. 87–100. Springer, Heidelberg (2010)
19. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann Publishers Inc. (1996)
20. Mallios, Y., Bauer, L., Kaynar, D., Ligatti, J.: Enforcing more with less: Formalizing target-aware run-time monitors. Tech. Rep. CMU-CyLab-12-009, CyLab, Carnegie Mellon University (2012)
21. Martinelli, F., Matteucci, I.: Through modeling to synthesis of security automata. Electron. Notes Theor. Comput. Sci. 179, 31–46 (2007)
22. Milner, R.: A Calculus of Communicating Systems. Springer (1982)
23. Schneider, F.B.: Enforceable security policies. ACM Trans. Inf. Syst. Secur. 3(1), 30–50 (2000)
24. Tuttle, M.R.: Hierarchical correctness proofs for distributed algorithms. Master's thesis, Dept. of Electrical Engineering and Computer Science. MIT (1987)