Enforcing More with Less: Formalizing Target-aware Run-time Monitors

Yannis Mallios, Lujo Bauer, Dilsun Kaynar, and Jay Ligatti

May 3, 2012

CMU-CyLab-12-009

CyLab
Carnegie Mellon University
Pittsburgh, PA 15213

Enforcing More with Less: Formalizing Target-aware Run-time Monitors

Yannis Mallios[2], Lujo Bauer[1,2], Dilsun Kaynar[1], and Jay Ligatti[3]

[1] CyLab, Carnegie Mellon University, Pittsburgh, USA
[2] ECE, Carnegie Mellon University, Pittsburgh, USA
{mallios,lbauer,dilsunk}@cmu.edu
[3] Dept. of Comp. Sci. and Eng., 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 study and explicitly formulate 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. Moreover, 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 for making 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 and mitigated. Example guarantees could be that programs: only access memory that has been allocated to them (memory safety); only jump to and execute valid code (control-flow integrity); use no more than 10 MB of storage and 10 KB/sec network bandwidth for grid use (resource allocation); 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 specific example of monitoring is system-call interposition (e.g., [27, 12]). Here, 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 by taking remedial action when a call violates the policy. This idea is depicted in Fig. 1. In practice, there are several instantiations of monitors for this general concept. Understanding and formally reasoning about specific instantiations is as important as understanding the general concept, since in enables us to reason about important details 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 being restricted to just terminating the application (e.g., when the application 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)[4].

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., [25, 19]) 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.

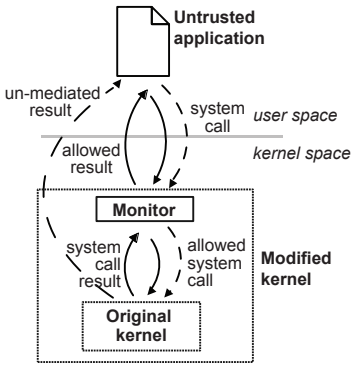


Fig. 1: System-call interposition: dashed line shows an input-mediating monitor; solid line an input/output-mediating monitor.

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 express the communication between the untrusted application, the monitor, and the kernel in a model, it might not be possible to differentiate between and compare monitors that can mediate all security-relevant communication between the application and the kernel (solid lines in Fig. 1) from monitors that can mediate only some of it (dashed lines in Fig. 1).

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

Some recent models (e.g., [20, 13]) make progress towards such more detailed reasoning by including in the model bi-directional communication between the monitor and its environment (e.g., application and kernel), but they 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 must be eventually closed cannot, in general, be enforced by any monitor, because the monitor does not know what the untrusted application will do in the future, and thus such a policy is outside its enforcement capabilities. 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 [22, 21] is a labeled transition model for asynchronous concurrent systems. Thus, we are using an automata-based formalism, similarly to many previous models of run-time enforcement mechanisms, with enough expressive power to model asynchronous systems (e.g., the 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, rather than assuming complete mediation [16, 5]. (We discuss more such examples in §3.)

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 types of monitors and 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 that previous models often reasoned about 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.

Roadmap. We start by briefly reviewing I/O automata (§2). We then informally show how to model monitors and targets in our framework and discuss some of the benefits of this approach (§3). Next, we formally define policies and enforcement (§4). Then, we show several examples of the meta-theoretical analysis that our framework enables by (a) providing some lower bounds for enforceable policies (§5), and (b) exposing constraints under which seemingly different monitoring architectures can enforce the same classes of policies (§5.3).

2 I/O Automata

I/O automata are a labeled transition model for asynchronous concurrent systems [22, 21]. In this section we informally review aspects of I/O automata that we build on in the rest of the paper. A more formal presentation can be found in App. A. 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*** $sig(A)$ of A. The signature $sig(A)$ is a triple of disjoint sets—$input(A)$, $output(A)$, and $internal(A)$. We write $acts(A)$ for $input(A) \cup output(A) \cup internal(A)$. We sometimes refer to output and internal actions as *locally-controlled* actions.

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, $trans(A) \subseteq states(A) \times acts(A) \times states(A)$, with the property 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 a countable number of equivalence classes.

If A has a transition (q, a, q') then we say that action a is *enabled* in state q. 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, $S|X = \{(s|X) \mid s \in S\}$.

An automaton that models a complex system can be constructed by *composing* automata that model the system's components. 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 components automata and its behaviors are the interleavings of the behaviors of the component automata, modulo the communication (synchronization on shared input-output actions); definitions of compatibility and some other technical details are given in App. A. Similarly to the composition of automata is defined the composition of modules [26].

Unlike in models such as CCS [24], composing two automata that share some actions (i.e., outputs of one automaton may be inputs to the other) causes those actions to be regarded as output actions of the composition. Those that are required to be internal need to be explicitly classified as such using the *hiding* operation. The operation of *renaming*, on the other hand, changes the names of actions, but not their types.

3 Specifying Targets and Monitors

We model targets (the entities to be monitored) and monitors as I/O automata. We let the metavariables $\mathcal{T}$ and $\mathcal{M}$ range over targets and monitors. Targets composed with monitors are called *monitored targets* or *monitored applications*; examples are the modified kernel and the safe application in Fig. 1. A monitored target might itself be a target for another monitor.

Building on the example of system-call interposition in Fig. 3, we now show how monitors and targets can be modeled using I/O automata. Suppose that 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 make a 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 the 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 monitors models, as identified also by more recent frameworks (e.g., [20, 13]).

Fig. 3a shows I/O automata interface diagrams of the monitored system consisting of the application and the monitored kernel.An I/O automaton definition for this kernel is shown in Fig. 2, using the standard precondition-effect style of writing transition relations for I/O automata.

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

Signature: Input: *OpenFile*(*fn*), where *fn* is a *file_name*
(type *file_name* = nat)
WriteFile(*fd*, *bytes*), where *fd* is a *file_descriptor*
(type *file_descriptor* = nat)
(type *bytes* = nat)
CloseFile(*fd*), where *fd* is a *file_descriptor*
Output: *Kill*(), *FD*(*fd*,*fn*), where *fd* is a *file_descriptor*
(type *file_descriptor* = nat)
and *fn* is the corresponding *file_name*.

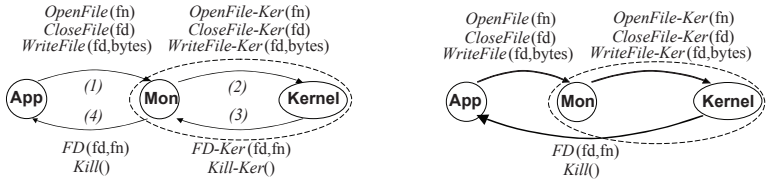
States: *req_list* : List of elements of type *file_name*
assigned_list : List of elements of type *file_descriptor*
kill : flag of type *bool*

Start States: *req_list* = *nil*
assigned_list = *nil*
kill = *false*

Transitions: *OpenFile*(*fn*)
Effect: *req_list* = *req_list*@[*fn*]
CloseFile(*fd*)
Effect: *assigned_list* = *assigned_list*\[*fn*], where
(*assigned_list**z*) denotes the function
that removes the element *z* from list *assigned_list*
WriteFile(*fd*, *bytes*)
Effect: Some lower level specification of write for writing
bytes on the actual file
FD(*fd*,*fn*)
Precondition: ¬*empty*(*req_list*) and ∃(*fn* : *file_name*) ∈ *req_list*,
where *empty* is a predicate on lists that returns *true* whenever
its argument is an empty list
Effect: *req_list* = (*req_list**fn*), where (*req_list**fn*) denotes the
function that removes the element *fn* from list *req_list*
assigned_list = *assigned_list*@[*fd*]
Kill()
Precondition: *kill* = *true*
Effect: *req_list* = *nil*
assigned_list = *nil*
and *kill* = *false*.

Fig. 2: Kernel I/O automaton definition

through the renamed actions of the kernel (using the renaming operation of I/O automata, §2): 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 by adding hooks that allow the monitor to intercept the target's actions. In practice, this is often accomplished by rewriting the target in order to inline or interpose a monitor. Finally, we also *hide* the communication between monitor and the kernel so that it remains internal to the monitored



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

Fig. 3: I/O automata interface diagrams of kernel, application, and monitor

target (denoted by the dotted line around the monitored kernel automaton). This is because we model 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 from §1 we described some choices that a monitor designer can make, such as choosing the (1) interface to be monitored, e.g., mediate only input actions, and (2) the trace modification capabilities of the monitor. We next describe how to express the above choices in our model.

Modeling the Monitored Interface. By appropriately restricting the renaming function applied to the target, we can model different monitoring architectures (e.g., input sanitization, §1). For example, in Fig. 3b, we renamed only the input actions of the kernel (i.e., *OpenFile*, *CloseFile*, and *WriteFile*). This allows us to model monitors that mediate inputs sent to the target and can prevent, for example, SQL injections attacks. Similarly, renaming only the outputs of the target we can model monitors that mediate only output actions (and can prevent, for example, cross-site scripting attacks).

Modeling Implementation Aspects of Monitors. When defining monitors of different enforcement capabilities, as the ones of previous models (e.g., [25, 18]), one can realize that mapping transition functions of previous models to transition relations of I/O automata does not suffice to uniquely identify practical implementations of monitors. The added expressiveness of I/O automata allows us to model different implementations of monitors that might make different choices about: whether the monitor can edit input before forwarding it to the target application; the extent to which the monitor can ignore the application; and the extent to which the monitor can use the application as an oracle or simulator to discover, in a controlled way, how it would respond to different input actions.

However, if we focus on a uni-directional communication path from the target to the monitor to the environment, then all types of monitors defined in previous work are expressible in our framework, e.g., *security automata* [25] (or *truncation automata* [19]), which halt targets that try to execute actions that violate the security policy; *suppression automata* [18], which can ignore some actions that the target wants to execute; and *edit automata* [19], which can both insert and remove actions from the trace that the target is producing.

Returning to our example: to model a truncation monitor that halts the kernel once the kernel outputs an *FD-Ker*(*fd*,*fn*) with an already-assigned file descriptor *fd*, we add a transition to the monitor that, upon receipt of the "bad" action, takes the monitor to a specific "halt state". Since the monitor is input enabled, that transition can be made regardless of which state the monitor is in. Once the monitor goes into this halt state, the only enabled output action will be a "halt" action to kill the kernel. The kernel will need to have this "halt" action as an input action, and an appropriate transition to stop its execution. Since the monitor is input enabled, it may, even when in the halt state, still receive invalid actions from the kernel until the kernel is halted. In previous models, for any action that the target wanted to execute, the target would wait for the monitor to finish considering that action before trying to execute the following one; in other words, the target and the monitor were synchronized. However, in our framework the monitor does not in general have such control over the target. These issues affect the policies that are enforceable by the monitor, since the target might try to execute a series of invalid actions before the monitor gets a change to take corrective action; we will revisit this point in the next section.

More general, a translation of monitors from previous monitors to ours must involve three steps that account for the added expressiveness of I/O automata. The first two are "definitional", i.e., they are related to the differences between the corresponding automata definitions. First, we need to extend the transition function of an automaton with transitions that model the input enabledness of the I/O automata. Second, in truncation, suppression, and edit automata the actions that the target was sending to the monitor belonged to the same action set as the ones that the monitor was forwarding to the environment. However, in I/O automata, the signature prohibits that, since the input and output actions must belong to disjoint sets. To account for that we need to define a bijection that will map the inputs of the truncation automata to fresh output actions. The third step involves the implicit assumptions made by previous models and exposed by our framework: a run-time monitor might not be able to control, in practice, how the target produces actions. More specifically, in previous models, for any action that the target wanted to execute, the monitor could decide and (perhaps) forward some action to the environment, and the target would wait for the monitor to finish before trying to execute the next action. In other words, the target and the monitor were synchronized. However, in our framework the monitor does not have such a control over the target. This means that the target might be producing actions without waiting for the monitor to make synchronized decisions. For that reason, our monitors need to have some data structure (e.g., a queue) to buffer the inputs from the target, and then, when given the chance (by fairness assumptions, for example), dequeue the corresponding actions and take appropriate action. Similarly, for truncation automata, the monitor might not have the ability to halt the target, unless we know that the target has some *halt* input action that will guarantee its termination (as in our system call interposition, with the *kill* system call). Next, we provide a transla-

tion of truncation automata [19] to I/O automata, to illustrate the above steps. Translations of other types of monitors can be defined similarly.

We will assume that the target can be terminated by a *stop* action. Given a truncation automaton $A_T = \langle Q, Q_0, \delta \rangle$ that is defined over some action set Σ_{A_T}, we define a truncation monitor $M_T = \langle sig(M_T),\ states(M_T),\ start(M_T),\ R_{M_T},\ Tasks(M_T) \rangle$, where:

1. $sig(M_T) = \langle input(M_T), internal(M_T), output(M_T) \rangle$, where:
   (i) $input(M_T) = \Sigma_{A_T}$,
   (ii) $internal(M_T) = \emptyset$,
   (iii) $output(M_T) = f(input(M_T)) \cup \{stop\}$,
   where $f : input(M_T) \xrightarrow[onto]{1-1} (\Sigma \backslash input(M_T))$.
2. $states(M_T) = (Q \times (input(M_T))^\star) \cup \{\langle halt, \epsilon \rangle\}$, i.e., the state of automaton together with the queue to buffer inputs from the target, plus an additional halt state,
3. $start(M_T) = Q_0 \times \{\epsilon\}$,
4. $trans(M_T) =$
   $\{\langle \langle q, \sigma \rangle, \iota, \langle q, \sigma; \iota \rangle \rangle \mid \langle q, \sigma \rangle \in states(M_T)$ and $\iota \in input(M_T)\}$
   $\cup\ \{\langle \langle q, \alpha; \sigma \rangle, f(\alpha), \langle q', \sigma \rangle \rangle \mid \langle q, \alpha; \sigma \rangle \in states(M_T)$ and $\delta(q, \alpha) = q'\}$
   $\cup\ \{\langle \langle q, \alpha; \sigma \rangle, stop, \langle halt, \epsilon \rangle \rangle \mid \langle q, \alpha; \sigma \rangle \in states(M_T)$ and $\delta(q, \alpha) = halt\}$,
5. Each action in $local(M_T)$ defines a unique equivalence class.

4 Policy Enforcement

In this section we define security policies and introduce two definitions of enforcement. The first defines enforcement with respect to a specific target, thus capturing scenarios in which the designer knows where the monitor is being installed (e.g., installing a system call interposition monitor to a specific version of a Linux kernel). The second one defines enforcement independently of the target, thus capturing scenarios in which the monitor designer might not know apriori the targets to which the monitor will be applied (e.g., when designing a system call interposition that enforces policies independently of the underlying kernel).

4.1 Security Policies

A *policy* is a set of (execution, schedule, or trace[5]) 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., [25, 19]) 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

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

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 will be 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.

Since I/O automata can have infinite states and produce possibly infinite computations, we would like to avoid arguments and discussions about computability and complexity issues: they are outside the scope of this paper. Thus, we make the assumption that all policies $\mathscr{P}$ that we discuss are *implementable* [26], meaning that for each module $\hat{P}$ in $\mathscr{P}$ under discussion, 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 and the target from leaking outside the composition[6]. If a target does not need renaming, rename can be the identity function; if we do not care about hiding all communication, the hiding function can apply to only some actions. For example, suppose the monitored target from our running example (node with

[6] Since Def. 1 reasons about schedules (i.e., internal actions as well as input and output), hide_Φ is redundant. We include it in this definition to expose the re-writing process that needs to happen for run-time enforcement in practical scenarios, but we will omit it in the rest of the paper.

dotted lines in Fig. 3b) 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[7]. 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 instances of Def. 1 and Def. 2 can be obtained by replacing schedules with traces (trace enforcement), by 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 with that of the policy [20, 5]. In this paper we focus on the subset and the equality relations, and refer to the corresponding enforcement definitions, respectively, as *sound* (e.g., Def. 1) and *precise enforcement*.

4.3 Comparing Enforcement Definitions

As a first example of meta-theoretic analysis in our framework, we compare these two definitions. More specifically, one might expect target-specific monitors to have an advantage in enforcement. If we have a monitor that enforces a policy for any target (i.e., a generalized monitor) then the monitor also specifically enforces the policy for some target $\mathcal{T}$. However, a monitor that is "customized" for enforcing a policy on a specific target (e.g., Linux) might not enforce the policy on any target (e.g., Windows).

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

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* $\mathcal{M}$).

[7] Monitors of previous models, such as [25] and [19], are generalized monitors.

Proofs are given in App. C. The proofs use several key I/O automata theorems, such as associativity of composition, or that renaming of compatible components can be done before or after composition. These theorems are included in App. B.

Prop. 1 compares the two definitions of enforcement (Def. 1 and Def. 2) with respect to the same monitor and shows that our definitions capture the intuitive notions of enforcement, i.e., a monitor that enforces a policy without being tailored for a specific target should enforce the policy on any target, while the inverse should not be true 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 left direction of the theorem is straightforward: any generalized monitor can be used as a target-specific monitor. The right direction is more interesting since it suggests, perhaps surprisingly, that it is possible to construct a generalized monitor from a target-specific one. More specifically, once we have a monitor that enforces a policy on a specific target, we can use this *monitored target* as the basis for a monitor on any other target. In that case, the only security-relevant behavior of the system would be exhibited by the monitor (formally, every action in every other target would be renamed to become security irrelevant). For example, suppose we have different versions of a specific application 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 simply 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 might be relevant when reinstalling the patched version of the application on top of other versions is more cost-efficient than finding patches for every other version.

Thm. 1 holds because Def. 1 and 2 place no restrictions on renaming functions (i.e., how a monitor is integrated with a target). In practice, this interaction may be more constrained. Thus, one might argue that it would be more natural to have the only-if direction of the theorem fail, since it erases the distinction between target-specific and generalized enforcement. This happens only if we restrict some elements of our definition of enforcement. For example, a constraint that can erase that distinction are presented in the following theorem.

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:*

$$\textit{(C1)}\quad acts(\hat{P}) \cap acts(\mathsf{rename}(\mathcal{T}')) \neq \emptyset,\ \textit{or}$$
$$\textit{(C2)}\quad range(\mathsf{rename}) \subseteq acts(\mathcal{M}')$$
$$\textit{and}\ (scheds(\mathcal{M}' \times \mathsf{rename}(\mathcal{T}'))|acts(\hat{P})) \subseteq scheds(\hat{P}))\Big).$$

Note that in the last argument we assumed that although the monitors are universally quantified they are still under our control, always trying to enforce the property. I.e., the quantifier does not range over arbitrary monitors that do not enforce the property if given the "opportunity" (such as exhibiting invalid actions).

The first condition prohibits us from finding an element in the policy that is irrelevant to the target that we are trying to monitor. For example, if we have two modules one containing networking events (i.e., the signature of the module contains only network related actions), and another one containing 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 ensures that the only way that we rename the target is to match it with some interface of the monitor. In other words, we do not arbitrarily rename the target, so that nobody can "listen" to its actions.

Although Thm. 2 enumerates a number of constraints under which the equivalence between generalized enforcement and target specific enforcement fails, no single restriction applies to all scenarios: for each constraint that erases the distinction between target-specific and generalized enforcement, one can find possible practical scenarios of policies and enforcement that the theorem should fail, but because the constraint is not satisfied it becomes trivially true. In other words, it seems (we conjecture) that there is no single set of constraints that allows Thm. 2 to be universally quantified over all policies and targets, as Thm. 1 is. Since our goal is to introduce a framework that is general enough to accommodate as many practical scenarios as possible (even seemingly degenerate ones), we rely on the monitor designer to impose appropriate restrictions on renaming or monitors to better reflect on the (practical) monitors under scrutiny.

5 Bounds on Enforceable Policies

The definitions and abstractions described thus far enable rigorous, detailed analyses of practical monitored systems, and also facilitate meta-theoretic reasoning that furthers our understanding of general limitations of practical monitors that fit this model. In this section we derive several such meta-theoretic results.

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 might argue that input-enabledness may lead to better design of systems because one has to consider all inputs that may be received from the

environment [21]. On the other hand, this constraint might be too restrictive for practical systems [1].

In our context, we believe that input-enabledness is a useful characteristic, since run-time monitors are by nature input-enabled systems: the 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 several other constraints to describe the lower bound of enforceability in our setting. The formal definitions of these constraints can be found in App. B.

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 that are typically absent from models of run-time enforcement. Our model, instead of making assumptions that might not hold in every practical scenario, e.g., that all actions can be mediated, takes a more nuanced view, which admits that there are aspects of enforcement outside the monitor's control, such as security-relevant actions that the monitor cannot observe or mediate (labeled as internal to a target), or the existence (or lack of) scheduling strategies that might not favor the monitor. The definitions above help us explicate these assumptions when reasoning about enforceable policies, as we see next.

5.2 Lower Bounds of Enforceable Policies

Another constraint that affects the lower bounds of enforceability and is semantically specific to monitoring is that, in practice, a monitored system cannot always ignore *all* behavior of the target application. Some realistic 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

the behaviors that the target can exhibit even if it receives no input. We call these policies *quiescent forgiving* (recall the definition of a quiescent state from §2). Modules contained is such policies are also called quiescent forgiving. This definition captures one type of limitation that was understood to be present in run-time monitoring, but that typically was not formally expressed. Quiescent forgiving modules can be defined more formally as follows:

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* $e = q_0, a_1, \ldots, q_n$:
$\Big(q_n \in quiescent(\mathcal{T}) \wedge \big(0 \leq i < n : q_i \notin quiescent(\mathcal{T})\big)\Big) \Rightarrow$
$\big(sched(e)|acts(\hat{P})\big) \in scheds(\hat{P}) \wedge (\forall i \in \mathbf{N} : 0 \leq i < n : (sched(q_0, \ldots, q_i) \mid acts(\hat{P})) \in scheds(\hat{P}))$.

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

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). Thus, in our context, even the equivalent of an edit monitor cannot enforce renewal properties (as opposed to [19]), since when renewal properties are constrained by prefix closure they collapse to safety. This is because, as mentioned above, our model of executions allows computation to stop at any point. We believe that this is another helpful characteristic of our model because it highlights that, in general, the system might stop executing for reasons that we cannot control, e.g., a power outage. In contrast, previous models (e.g., [19]) assumed that enabled actions of a monitor would always be performed. In our framework, any similar guarantees about the runs of the system are not built into the basic definitions but can be explicitly added through fairness and other similar constraints on I/O automata [17]. This is another instance of our framework making explicit the (practical) assumptions and constraints that affect the enforcement of policies. Earlier results about the enforcement powers of different types of monitors (e.g., that truncation monitors enforce safety policies 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. Hence, if a monitor composed with an application is to exhibit the same range of behaviors as the unmonitored application, it will have to consult the target application in order 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 an input forwarded by the monitor to an application might cause the application to execute internal or output actions (e.g., a buffer overflow) that are not allowed by the policy and that the monitor cannot prevent, since these are outside of 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, even if it does not have access to its state. This knowledge can be exploited to use simpler-than-expected monitors to enforce of (seemingly) complex policies. Although similar observations have been made before (e.g., program re-writing [15], non-uniformity [19], use of static analysis [7]), our framework can be used to formally extend them, as we demonstrate in the following section.

5.3 Policies Enforceable by Target-specific Monitors

As discussed in §3, the expressiveness of our model allows multiple ways to define monitors, e.g., a truncation monitor, that had a single natural definition in previous models. Due to space limitations, rather than comprehensively analyzing the policies enforceable by specific monitors, as done in previous work [25, 18–20, 15], we demonstrate two instances in which our framework enables formal results that can be exploited by designers of run-time monitors who have knowledge about the target application. The first (described in the remainder of §5.3) is a novel analysis of how some knowledge of the target can compensate (in terms of enforceability) for a narrower monitoring interface. The second focuses on the trace-modification capabilities of monitors and illustrates how arguments that were difficult to formalize in less expressive frameworks (e.g., [19]) can be naturally discussed in our model.

In order to formalize this statement in our framework, we will first define what it means for a monitored target to be input/output mediating and input mediating. The definitions formalize the constraints on the renaming functions of the monitored target, as they were described in Section 3.

Definition 6 *A monitor $\mathcal{M}$ is input/output mediating iff: $\forall \mathcal{T}$: $\forall$*rename*:*

1. $output(\mathsf{rename}(\mathcal{T})) \subseteq input(\mathcal{M})$
2. $input(\mathsf{rename}(\mathcal{T})) \subseteq output(\mathcal{M})$
3. $internal(\mathsf{rename}(\mathcal{T})) = internal(\mathcal{T})$
4. $output(\mathcal{T}) \subseteq output(\mathcal{M})$
5. $input(\mathcal{T}) \subseteq input(\mathcal{M})$

Constraints (1-3) force the renaming function to match the interfaces of the target and the monitor, i.e., it does not allow to arbitrarily rename the target interface, and ensure that all security relevant input/output behavior of the target is completely mediated by the monitor. In particular, constraint (1) ensures that all security relevant outputs will be received by the monitor, while constraint (2) ensures that all the security relevant inputs to the target will come from the

monitor. Constraint (3) ensures that the security relevant actions of the target are not renamed so that we can capture the fact that there are actions that are outside the monitor's control: if we could rename them to some internal actions of the monitor, then since the monitor controls its own internal actions, it would be possible to not exhibit invalid internal actions. Finally, constraints (4) and (5) ensure that the monitor has the ability to input and output the actions that the original target could.

Similarly to Definition 6 we can define a monitored target to be *input mediating*:

Definition 7 *A monitor $\mathcal{M}$ is input mediating iff: $\forall \mathcal{T}$: $\forall$*rename*:*

1. $input(\mathsf{rename}(\mathcal{T})) \subseteq output(\mathcal{M})$
2. $internal(\mathsf{rename}(\mathcal{T})) = internal(\mathcal{T})$
3. $output(\mathsf{rename}(\mathcal{T})) = output(\mathcal{T})$
4. $input(\mathcal{T}) \subseteq input(\mathcal{M})$

Constraint (1) ensures that all the security relevant inputs to the target will come for the monitor. Constraints (2) and (3) ensure that the security relevant actions of the target are not renamed so that we can capture the fact that there are actions that are outside the monitor's control, this time including the output actions of the target. Finally, constraint (4) ensures that the monitor has the ability to input all the actions that the original target could.

In §3 we described two monitoring architectures: one in which the monitor mediates the inputs and the outputs of the target, and another in which 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, there exist policies that are enforceable by input/output mediating monitors, but not by input mediating monitors. This can be expressed as follows:

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 $\mathcal{M}_2$), if the policy does not reason about the communication between the monitor and the target[8].

The constraint in Thm. 4 identifies those policies for which input/output mediating architectures are at least as powerful as input mediating architectures.If the policy reasons about, and prohibits, (some) communication between the target and the monitor, then the two architectures are equivalent.

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

[8] If we used trace enforcement, the constraint would be superfluous. We used schedules to remain consistent with other theorems in the paper.

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. The following theorem characterizes the targets for which this equivalence holds.

The theorem focuses on targets that previous research of run-time enforcement (e.g., [15, 19]) has focused on, which are expected to run, and interact with their environment, for infinitely long, such as operating systems and graphical user interfaces. Moreover, it assumes monitors that have both static and dynamic knowledge about the target, i.e., the monitor knows the future possible behaviors of the target, along with the state that the target is currently in.

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*[9]*,*
- ***(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*
- ***(C4)*** *$\forall e \in execs(\mathcal{T})$: if e has two quiescent states separated by a sequence of local actions, and any prefix of e ending between those states violates $\mathscr{P}$, then all prefixes of e ending between those states violate $\mathscr{P}$ and the two quiescent states are the same.*

Constraint **C2** ensures that whatever the target chooses to output in the beginning of its execution, and until it blocks for some input, obeys the policy. Constraint **C3** ensures that the input/output-mediating monitor does not have an "unfair" advantage over the input mediating one, just because the policy requires from the monitor to output actions, even if the target would never perform them. Constraint **C4** ensures that whenever the target receives some input (from the monitor), then no behavior that it exhibits (until it blocks to wait for another input) will violate the policy, or, if it does, then that behavior can be suppressed without affecting the target's future behavior.

The above theorem characterizes equivalence of enforcement for an important class of targets and monitors. However, if we want to reason about the general case, i.e, targets that might not execute "forever" and might not interact with their environment infinitely long, and monitors that do not have any

[9] As before, if we used trace enforcement, the constraint would be superfluous; thus the constraint is more of a technical issue rather a key idea in the theorem. We used schedules to remain consistent with other theorems in the paper.

dynamic information about the (execution of) target, then the following theorem characterizes the constraints for which the equivalence holds.

Theorem 6. $\forall \mathscr{P} : \forall \mathcal{T}$:
$\mathscr{P}$ 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)*** $\forall \hat{P} \in \mathscr{P} : scheds(\hat{P}) \subseteq scheds(\mathcal{T})$*, and*
- ***(C4)*** $\forall \hat{P} \in \mathscr{P}$*:* $\forall s \in scheds(\mathcal{T})$:
  $s \notin scheds(\hat{P}) \Rightarrow \exists s' \preceq s$:
  $(s' \in scheds(\hat{P})) \wedge (s' = s''; a) \wedge (a \in input(\mathcal{T}))$
  $\wedge$ $(\forall t \succeq s'$*:* $t \in scheds(\mathcal{T}) \Rightarrow t \notin scheds(\hat{P}))$.

Thm. 5 and Thm. 6 are an illustration of how our framework can help in making sound decisions for designing and implementing run-time monitors in practice. For example, suppose we have a Unix kernel and want to enforce the policy that that secret file cannot be (a) deleted or (b) displayed to guest users. A monitor designer who wants to *precisely* enforce that policy 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), because it does not know whether the kernel can, for example, correctly identify guest users and not display secret files to them. However, the designer can check if the specific kernel meets the constraints of Thm. 5. If it does, e.g., the kernel does not display any secret files while booting (i.e., **C2**), and does not display secret files to guest users, e.g., through a correct access-control mechanism (i.e., **C4**), then an input-mediating monitor suffices to enforce the policy. The correctness of such design choices might not always be obvious, and the above example demonstrates how our framework can aid in making more informed decisions. Moreover, such decisions can have benefits both in efficiency (by not monitoring the kernel's output sequence at run time), and in security (since the TCB/attack surface of the monitor is smaller).

Similarly to identifying constraints under which seemingly different monitoring architectures become equivalent given some knowledge about the intended target, we can identify constraints under which monitors of different enforcement capabilities are equivalent.

Thm. 7 explicates the constraints under which a target-specific truncation monitor is equivalent to a target-specific edit monitor. We remind the reader that these monitors are not generally equivalent [19]. Before presenting the theorem, we will provide the proof of a lemma, which formalizes in our formalism that truncation monitors can enforce safety properties. This lemma is being used in the proof of Thm. 7.

Lemma 1. *Given a schedule module $\hat{P}$ and a target $\mathcal{T}$, then there exists a truncation monitor $\mathcal{M}_T$ such that $\hat{P}$ is specifically precisely enforceable on $\mathcal{T}$ by $\mathcal{M}_T$ if:*

- ***(C1)*** $\forall \hat{P} \in \mathscr{P} : scheds(\hat{P}) \subseteq scheds(\mathcal{T})$,
- ***(C2)*** $\forall t \in scheds(\mathcal{T}) :$
  $t \notin scheds(\hat{P}) \Rightarrow \exists s \preceq t :$
  $\forall k \in (acts(\mathcal{T}) \cup acts(\hat{P}))^{\omega} : s \prec k :$
  $k \in scheds(\mathcal{T}) \Rightarrow k \notin scheds(\hat{P})$, *and*
- ***(C3)*** *$\hat{P}$ does not reason about the communication between the target and the monitor and the internal actions of the target.*

Now, we present the theorem that explicates the constraints under which a target-specific truncation monitor is equivalent to a target-specific edit monitor.

Theorem 7. *$\forall \mathscr{P}$: $\forall \mathcal{T}$: $\mathscr{P}$ is specifically precisely enforceable on $\mathcal{T}$ by some truncation monitor $\mathcal{M}_1$ iff $\mathscr{P}$ is specifically precisely enforceable on $\mathcal{T}$ by some edit monitor $\mathcal{M}_2$, if: $\forall \hat{P} \in \mathscr{P}$:*

- ***(C1)*** $\forall \hat{P} \in \mathscr{P} : scheds(\hat{P}) \subseteq scheds(\mathcal{T})$,
- ***(C3)*** $\forall \hat{P} \in \mathscr{P}$: $\forall t \in scheds(\mathcal{T}) :$
  $t \notin scheds(\hat{P}) \Rightarrow \exists s \preceq t :$
  $\forall k \in (acts(\mathcal{T}) \cup acts(\hat{P}))^{\omega} : s \prec k :$
  $k \in scheds(\mathcal{T}) \Rightarrow k \notin scheds(\hat{P})$, *and*
- ***(C3)*** *$\hat{P}$ does not reason about the communication between the target and the monitor and the internal actions of the target.*

6 Related Work

The first model of run-time monitors, *security automata*, was based on Büchi Automata and introduced by Schneider [25]. Since then, several similar models have been proposed that extend or refine the class of enforceable policies based on the enforcement and computational powers of monitors (e.g., [19, 11, 4, 10, 3]). Contrary to these models, we focus on modeling, in addition to the monitors, 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 of the previous frameworks [20].

Hamlen et al. described a model based on Turing Machines [15], with which they compared the classes of policies enforceable by several types of enforcement mechanisms, such as static analysis and inlined monitors. The main differences between this model with ours is that we model explicitly the communication of the monitor with the target and the environment, and we do not consider monitoring through rewriting the target application.

Recent work has revised these models or adopted alternate ones, such as the Calculus of Communicating Systems (CCS) [24] 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 revising existing models is Ligatti and Reddy's Mandatory Results Automata, which model the (synchronous) communication between the monitor and the target [20]. MRA's, however, do not model the target explicitly, and thus results about enforceable policies in target-specific environments might be difficult to derive.

Among the works building on CCS or CSP is Martinelli and Matteucci's model of run-time monitors based on CCS [23]. 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 [2]. This work focuses on the synchronization between a single monitor and target application, although the language is expressive enough to capture many other enforcement scenarios. Our work is similar to Basin's, however we focus more on showing how such a more expressive framework can be used to derive meta-theoretical results on enforceable policies in different scenarios, instead of focusing on the (complementary aspect) of showing how to faithfully translate and model practical scenarios in such frameworks.

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

7 Conclusion

Formal models for run-time monitors have helped improve our understanding of the powers and limitations of enforcement mechanisms [25, 19], and aided in their design and implementation [9, 14]. However, these models often fail to capture many details and complexity relevant to real-world run-time 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 of run-time monitoring. Moreover, we show how this framework can be used for meta-theoretic analysis of enforceable security policies. In particular, we derive results that describe lower bounds on enforceable policies that are independent of the particular choice of monitor (Thm. 3). We also identify constraints under which monitors with different monitoring and enforcement capabilities (i.e., monitors that see only a subset of the target's actions; and monitors that have more or less ability to correct a target's invalid behavior) can enforce the same classes of policies (Thm. 5 and 7).

References

1. de Alfaro, L., Henzinger, T.A.: Interface automata. In: Proceedings of the 8th European software engineering conference. pp. 109–120 (2001)
2. Basin, D., Olderog, E.R., Sevinc, P.E.: Specifying and analyzing security automata using CSP-OZ. In: Proceedings of the 2nd ACM symposium on Information, computer and communications security. pp. 70–81. ASIACCS '07 (2007)
3. Basin, D.A., Jugé, V., Klaedtke, F., Zalinescu, E.: Enforceable security policies revisited. In: POST. pp. 309–328 (2012)
4. Bielova, N., Massacci, F.: Do you really mean what you actually enforced? International Journal of Information Security pp. 1–16 (2011)
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 (June 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: IEEE Computer Security Foundations Symposium (2008)
9. Erlingsson, U., Schneider, F.B.: SASI enforcement of security policies: a retrospective. In: Workshop on New security paradigms (2000)
10. Falcone, Y., Fernandez, J.C., Mounier, L.: What can you verify and enforce at runtime? International Journal on Software Tools for Technology Transfer (STTT) pp. 1–34 (2011)
11. Fong, P.W.L.: Access control by tracking shallow execution history. In: Proceedings of the 2004 IEEE Symposium on Security and Privacy. pp. 43–55 (2004)
12. Garfinkel, T.: Traps and pitfalls: Practical problems in in system call interposition based security tools. In: Network and Distributed Systems Security Symposium (2003)
13. Gay, R., Mantel, H., Sprick, B.: Service automata. In: 8th International Workshop on Formal Aspects of Security and Trust (2011)
14. Hamlen, K.: Security policy enforcement by automated program-rewriting. Ph.D. thesis, Cornell University (2006)
15. Hamlen, K.W., Morrisett, G., Schneider, F.B.: Computability classes for enforcement mechanisms. ACM Trans. Program. Lang. Syst. 28(1), 175–205 (2006)
16. H.Salzer, J., Schroeder, M.D.: The protection of information in computer systems. In: Fourth ACM Symposium on Operating System Principles (Mar 1973)
17. Kwiatkowska, M.: Survey of fairness notions. Information and Software Technology 31(7), 371 – 386 (1989)
18. Ligatti, J., Bauer, L., Walker, D.: Enforcing non-safety security policies with program monitors. In: European Symposium on Research in Computer Security (ESORICS). vol. 3679, pp. 355–373 (2005)
19. Ligatti, J., Bauer, L., Walker, D.: Run-time enforcement of nonsafety policies. ACM Transactions on Information and System Security 12(3) (2009)
20. Ligatti, J., Reddy, S.: A theory of runtime enforcement, with results. In: European Symposium on Research in Computer Security (ESORICS) (2010)
21. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann Publishers Inc. (1996)
22. Lynch, N.A., Tuttle, M.R.: Hierarchical correctness proofs for distributed algorithms. In: ACM Symposium on Principles of Distributed Computing (1987)

23. Martinelli, F., Matteucci, I.: Through modeling to synthesis of security automata. Electron. Notes Theor. Comput. Sci. 179, 31–46 (July 2007)
24. Milner, R.: A Calculus of Communicating Systems. Springer-Verlag New York, Inc. (1982)
25. Schneider, F.B.: Enforceable security policies. ACM Trans. Inf. Syst. Secur. 3(1), 30–50 (2000)
26. Tuttle, M.R.: Hierarchical correctness proofs for distributed algorithms. Master's thesis, Dept. of Electrical Engineering and Computer Science, MIT (1987)
27. Wagner, D.A.: Janus: an approach for confinement of untrusted applications. Tech. Rep. UCB/CSD-99-1056, EECS, University of California, Berkeley (1999)

Appendix A. I/O Automata Formal Definitions

We assume **universe** of program *actions* denoted as Σ.

Definition 8 *(Action Signature) An* ***action signature*** S *is a triple of three disjoint sets of actions. The disjoints sets of a signature are* ***input***, ***output***, *and* ***internal*** *actions (denoted as* $input(S)$, $output(S)$, *and* $internal(S)$*). The* ***external*** *actions* $external(S)$=$input(S) \cup output(S)$ *model the interaction of the automaton with the evironment.* $Local(S)$=$internal(S) \cup output(S)$ *are the* ***local*** *actions that are under the automaton's control.*

Definition 9 *(I/O automata) An* ***I/O automaton*** *is defined by the following components:*

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,* $trans(A) \subseteq states(A) \times acts(A) \times states(A)$*, with the property 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* $Local(A)$ *into at most a countable number of equivalence classes.*

Definition 10 *(Executions) An* ***execution*** e *of* A *is either a finite sequence,* $q_0, a_1, q_1, a_2, \ldots, a_r, q_r$*, or an infinite sequence* $q_0, a_1, q_1, a_2, \ldots, a_r, q_r, \ldots$ *of alternating states and actions such that* $(q_k, a_{k+1}, q_{k+1}) \in trans(A)$ *for* $k \geq 0$*.*

Definition 11 *(Schedules) A* ***schedule*** s *of an execution* e *of* A *denoted by* $\text{sched}(e)$*, is the subsequence of* e *consisting of only actions.*

Definition 12 *(Traces) A* ***trace*** t *of an execution* e *(or schedule* s*) of an I/O automaton* A*, denoted by* $\text{trace}(e)$ *(or* $\text{trace}(s)$*), is defined as the subsequence of* e *(or* s*) consisting of all the external actions.*

Definition 13 *(Execution module) An execution module E consists of a set $states(E)$, an action signature $sig(E)$ and a set $execs(E)$ of executions. We will denote the set of schedules of E with $scheds(E)$, and the set of traces or E with $traces(E)$. An execution module E is said to be an execution module of an automaton A if E and A have the same states, the same action signature, and the executions of E are a subset of the executions of A.*

Definition 14 *(Schedule module) A schedule module S consists of an action signature $sig(S)$ together with a set of schedules $scheds(S)$.*

Definition 15 *(Trace module) A trace module T consists of an action signature $sig(T)$ together with a set of traces $scheds(T)$.*

Definition 16 *(Parallel composition of I/O automata) When composing automata S_i, where $i \in I$, or modules, their signatures are called* compatible *if their output actions are disjoint and the internal actions of each automaton are disjoint with all actions of the other automata. More formally, The actions signatures $S_i : i \in I$ or called compatible if for all $i, j \in I$:*

1. $output(S_i) \cap output(S_j) = \emptyset$
2. $internal(S_i) \cap acts(S_j) = \emptyset$

When the signatures are compatible we say that the corresponding automata and modules are compatible too.

The composition $A = \prod_{i \in I} A_i$ of a set of compatible automata $\{A_i : i \in I\}$ is defined as:

1. $states(A) = \prod_{i \in I} states(A_i)$
2. $start(A) = \prod_{i \in I} start(A_i)$,
3. $sig(A) = \prod_{i \in I} sig(A_i) =$
   $\Big(output(A) = \cup_{i \in I} output(A_i)$,
   $internal(A) = \cup_{i \in I} internal(A_i)$,
   $input(A) = \cup_{i \in I} input(A_i) - \cup_{j \in I} output(A_j)\Big)$,
4. $trans(A)$ *is equal to the set of triples (q, a, q') such that for all $i \in I$*
   (a) if $a \in acts(A_i)$ then $(q_i, a, q_i') \in trans(A_i)$, and
   (b) if $a \notin acts(A_i)$ then $q_i = q_i'$
5. $Tasks(A) = \cup_{i \in I} Tasks(A_i)$

Definition 17 *(Parallel composition of modules) Similarly to the composition of automata, we can define the composition of execution modules. Specifically, given a countable collection of compatible execution modules $\{E_i, i \in I\}$ we define the composed execution module $E = \prod_{i \in I} E_i$ as follows. The states of E are $\prod_{i \in I} states(E_i)$, and the action signature $\prod_{i \in I} sig(E_i)$. Given a sequence $x = q_0 a_1 q_1 \ldots$ of states and actions of E we define $x|E_i$ to be the sequence obtained by removing $a_j q_j$ if a_j is not an action of E_i, and replacing the remaining q_j*

with $q_j|E_i$. The executions of E are those sequences $q_0a_1q_1\ldots$ such that for every $i \in I$ we have that $x|E_i$ is an execution of E_i, and that $q_{j-1}|E_i = q_j|E_i$ whenever a_j is not an action of E_i.

Given a countable collection of compatible schedule (or trace) modules $\{S_i, i \in I\}$ we define the composed execution module $S = \prod_{i\in I} S_i$:

1. *$sig(E) = \prod_{i\in I} sig(S_i)$,*
2. *$execs(E)$ is the set of executions s such that the subsequence s' of s consisting of actions of S_i, is a schedule of S_i for every $i \in I$.*

The I/O automata definition also includes the equivalence relation $Tasks(A)$. This is used in the definition of *fairness*, which essentially says that the automaton will give fair turns to each of its tasks while executing.

Definition 18 *(Fairness) A task C is enabled in a state q if some action in C is enabled in q.*

An execution e of an I/O automaton A is said to be fair *if for each class C of $Tasks(A)$: (1) if e is finite, then C is not enabled in the final state of e, or (2) if e is infinite, then e contains either infinitely many events from C or infinitely many occurrences of states in which C is not enabled.*

Fairness abstracts the need for modeling a scheduler in the system. Specifically, when reasoning about practical systems, instead of explicitly modeling a scheduler one can simply reason about a fair version of the system. The type of fairness that I/O automata define is called "weak fairness, and is only one of the many different types of fairness [17].

Given an automaton or a module A we denote the sets of fair executions, schedules and traces by $fairexecs(A)$, $fairscheds(A)$ and $fairtraces(A)$.

Definition 19 *(Hiding) If S is a signature and $\Phi \subseteq output(S)$, then $\mathsf{hide}_\Phi(S)$ is defined to be the new signature S', where $input(S') = input(S)$, $output(S') = output(S) - \Phi$, and $internal(S') = internal(S) \cup \Phi$. Given an I/O automaton A and $\Phi \subseteq output(A)$, $\mathsf{hide}_\Phi(A)$ is the automaton A' obtained by replacing $sig(A)$ with $sig(A') = \mathsf{hide}_\Phi(sig(A))$.*

Definition 20 *(Renaming) An* action mapping *(or renaming) f is a total injective mapping between sets of actions. Such a renaming is said to be* applicable *to an automaton if the domain of f contains the actions of the automaton. If the renaming f is applicable to an automaton A, then the automaton $\mathsf{rename}(A)$ is the automaton with the states and start states of A; with the input, output and internal actions $\mathsf{rename}(input(A))$, $\mathsf{rename}(output(A))$, $\mathsf{rename}(internal(A))$ respectively; with the transition relation $\{(q, \mathsf{rename}(a), q') : (q, a, q') \in trans(A)\}$; and the equivalence relation $\{(\mathsf{rename}(a), \mathsf{rename}(a')) : (a, a') \in tasksA\}$.*

Hiding and renaming for modules are defined similarly to Definition 19 and Definition 20.

The following definition considers how renaming interacts with composition. It reveals the constraints under which when composing renamed component automata, there is some behavior-preserving way to rename the composition of the components, and when two component automata communicate on some action a and we rename their composition, there exists a behaviorally equivalent composed automaton, in which the components are renamed.

Definition 21 *(Composition of renaming functions) A collection $\{f_i : i \in I\}$ of action mappings is* compatible *if for all actions a_i and a_j we have $f_i(a_i) = f_j(a_j)$ iff $a_i = a_j$.*

We define the composition of collection $\{f_i : i \in I\}$ of compatible action mappings to be the action mapping having as its domain the union of the domains of the f_i, and mapping the action a to $f_i(a)$ if a is in the domain of f_i. The fact that f_i are compatible ensures that f is well defined.

Appendix B. I/O Automata Theorems

This appendix contains the formal expression of a collection of several important I/O automata theorems. Their discussion and proofs can be found in [22, 26, 21].

Theorem 8. *An execution of a composition induces executions of the component automata*
Let $\{A_i\}_{i\in I}$ be a collection of compatible automata and let $A = \prod_{i\in I} A_i$. If $\alpha \in execs(A)$ then $\alpha|A_i \in execs(A_i)$ for every $i \in I$. Moreover, the same results holds for scheds(), traces() and their fair versions.

Theorem 9. *Executions of component automata can often be pasted together to form an execution of the composition*
Let $\{A_i\}_{i\in I}$ be a collection of compatible automata and let $A = \prod_{i\in I} A_i$. Suppose α_i is an execution of A_i for every $i \in I$, and suppose β is a sequence of actions in $acts(A)$ such that $\beta|A_i = sched(\alpha_i)$ for every $i \in I$. Then there is an execution α of A such that $\beta = sched(\alpha)$ and $\alpha_{=}\alpha|A_i$ for every $i \in I$. Moreover the same holds for external() and traces() instead of acts() and sched(), and for fair executions.

Theorem 10. *Schedules and traces of component automata can be pasted together to form schedules and traces of the composition*
Let $\{A_i\}_{i\in I}$ be a collection of compatible automata and let $A = \prod_{i\in I} A_i$. Let β be a sequence of actions in $acts(A)$. If $\beta|A_i \in scheds(A_i)$ for every $i \in I$, then $\beta \in scheds(A)$.Moreover the same holds for external() and traces() instead of acts() and sched(), and for their fair versions.

Theorem 11. *Composition of modules correspond to composition of automata*
Let $\{A_i\}_{i\in I}$ be a collection of compatible automata. Then $execs(\prod_{i\in I} A_i) = \prod_{i\in I} execs(A_i)$, $scheds(\prod_{i\in I} A_i) = \prod_{i\in I} scheds(A_i)$, $traces(\prod_{i\in I} A_i) = \prod_{i\in I} traces(A_i)$, $fairexecs(\prod_{i\in I} A_i) = \prod_{i\in I} fairexecs(A_i)$, $fairscheds(\prod_{i\in I} A_i) = \prod_{i\in I} fairscheds(A_i)$, $fairtraces(\prod_{i\in I} A_i) = \prod_{i\in I} fairtraces(A_i)$. Moreover the same hold for execution, schedule and trace modules.

Theorem 12. *Commutativity and Associativity of composition of single components*
Let A, B and C I/O automata. Then:

1. $A \times B = B \times A$
2. $(A \times B) \times C = A \times (B \times C) = A \times B \times C$

Theorem 13. *Commutativity, Associativity and Congruence of automata and modules*
Let $A = \prod_i A_i$, $B = \prod_i B_i$ and $C = \prod_i C_i$ and $D = \prod_i D_i$ where A_i, B_i, C_i and D_i are either I/O automata or modules. Then:

1. $A \times B = B \times A$
2. $(A \times B) \times C = A \times (B \times C) = A \times B \times C$
3. *if $A = B$ and $C = D$, then $A \times C = B \times D$ whenever $A \times B$ and $C \times D$ are defined.*

Theorem 14. *Hiding on Automata passes to modules*
For all automata A, execution modules E, schedule modules S and sets of actions Σ:

1. $execs(\mathsf{hide}_\Sigma(A)) = \mathsf{hide}_\Sigma(execs(A))$
2. $scheds(\mathsf{hide}_\Sigma(E)) = \mathsf{hide}_\Sigma(scheds(E))$
3. $traces(\mathsf{hide}_\Sigma(S)) = \mathsf{hide}_\Sigma(traces(S))$

Notice that the last two versions, also hold for automata besides execution and trace modules.

Theorem 15. *Hiding of composition corresponds to hiding of components*
Let $\{M_i : i \in I\}$ be a collection of compatible automata or modules, and let $\{\Sigma_i : i \in I\}$ be a collection of sets of actions. If $acts(M_i)$ and Σ_j are disjoint for all $i \neq j$ then: $\mathsf{hide}_{\cup_i \Sigma_i}(\prod_{i \in I} M_i) = \prod_{i \in I} \mathsf{hide}_{\Sigma_i}(M_i)$.

Theorem 16. *Renaming of composition corresponds to renaming of components*
Let $\{M_i : i \in I\}$ be a collection of compatible automata or modules, and let $\{\mathsf{rename}_i : i \in I\}$ be compatible action mappings. If rename_i is applicable to M_i for every $i \in I$, then $(\prod_{i \in I} \mathsf{rename}_i)(\prod_{i \in I} M_i) = \prod_{i \in I} \mathsf{rename}_i M_i$.

Theorem 17. *Renaming on Automata passes to modules*
Let rename *be a renaming applicable to the automaton A, execution module E and schedule module S:*

1. $execs(\mathsf{rename}(A)) = \mathsf{rename}(execs(A))$
2. $scheds(\mathsf{rename}(E)) = \mathsf{rename}(scheds(E))$
3. $traces(\mathsf{rename}(S)) = \mathsf{rename}(traces(S))$.

Theorem 18. *Sequence of hiding and renaming does not matter*
$\mathsf{hide}_{f(\Sigma)}(f(M)) = f(\mathsf{hide}_\Sigma(M))$ *for any automaton or module M and applicable renaming f.*

Appendix C. Proofs of Theorems

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

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 $\mathcal{M}$).

Proof. For *(1)*, we assume that we have an arbitrary $\mathscr{P}$ that is generally soundly enforceable by some $\mathcal{M}$. By Def. 2, this means that:

> for all targets $\mathcal{T}$ there exists a module $\hat{P} \in \mathscr{P}$, a renaming function rename, such that $(scheds(\mathcal{M} \times \mathsf{rename}(\mathcal{T}))|acts(\hat{P})) \subseteq scheds(\hat{P})$. **(A)**

We have to show that for all targets $\mathcal{T}'$, $\mathscr{P}$ is specifically soundly enforceable on $\mathcal{T}'$ by $\mathcal{M}$, which by Def. 1 means that we have ti show that for some arbitrary $\mathcal{T}'$ there exists a module $\hat{P}' \in \mathscr{P}$, a renaming function rename', such that $(scheds(\mathcal{M} \times \mathsf{rename}'(\mathcal{T}'))|acts(\hat{P}')) \subseteq scheds(\hat{P}')$.

By **(A)** we know that there are $\hat{P}$ and rename that correspond to any $\mathcal{T}$, and thus for $\mathcal{T}'$. Use the corresponding choices of $\hat{P}$ and rename for $\mathcal{T}'$ and our claim follows from **(A)** immediately.

For *(2)*, we must exhibit a $\mathscr{P}$ and a $\mathcal{T}$ such that $\mathscr{P}$ is **specifically soundly enforceable** on $\mathcal{T}$ by $\mathcal{M}$ and it is not the case that $\mathscr{P}$ is **generally soundly enforceable** by $\mathcal{M}$.

Let $\mathscr{P} = \{scheds(\mathcal{M}) \cup \{\langle a \rangle \mid a \in \Sigma - acts(\mathcal{M})\}\}$. Also, let $\mathcal{T}$ be the trivial automaton, i.e., the I/O automaton with the empty set for actions and just a single start state. Thus, $scheds(\mathcal{T}) = \{\epsilon\}$. It is easy to see that $\mathscr{P}$ is specifically soundly enforceable on $\mathcal{T}$ by $\mathcal{M}$, i.e., that there exists a module $\hat{P} \in \mathscr{P}$, a renaming function rename, such that $(scheds(\mathcal{M} \times \mathsf{rename}(\mathcal{T}))|acts(\hat{P})) \subseteq scheds(\hat{P})$. $\mathscr{P}$ contains only one element, so $\hat{P} = scheds(\mathcal{M}) \cup \{\langle a \rangle \mid a \in \Sigma - acts(\mathcal{M})\}$ which contains all schedules that $\mathcal{M}$ can produce. Moreover, let rename be the identity function. From Thm. 8 we know that $scheds(\mathcal{M} \times \mathsf{rename}(\mathcal{T}))$ will be the pasting of the schedules of the two components, and since the schedules of the component $\mathsf{rename}(\mathcal{T})$ is just the empty sequence, $scheds(\mathcal{M} \times \mathsf{rename}(\mathcal{T})) = scheds(\mathcal{M})$. So we have to show that $scheds(\mathcal{M})|acts(\hat{P})) \subseteq scheds(\mathcal{M}) \cup \{\langle a \rangle \mid a \in \Sigma - acts(\mathcal{M})\}$, which is trivially true.

To prove the second conjunct of the claim, i.e., that it is not the case that $\mathscr{P}$ is **generally soundly enforceable** by $\mathcal{M}$, pick any $\mathcal{T}'$ that has as a signature only one output action, and produces some finite sequence of repetitions of this action of length greater than 1; i.e., $scheds(\mathcal{T}') = \{(a;a)^n \mid n \geq 1 \text{ and } a \in output(\mathcal{T}')\}$. Note that no matter how we rename $\mathcal{T}'$, its renamed output actions will still be an action of $\hat{P}$, since we added all actions that are not actions of the monitor $(\Sigma - acts(\mathcal{M}))$. Using Thm. 8 again, we see that the schedules of the composition will contain schedules of the component $\mathsf{rename}(\mathcal{T}')$, which means that there

is some sequence $s = (a; a) \in scheds(\mathcal{T}')$, where $a \in acts(\mathsf{rename}(\mathcal{T}'))$. But $s \notin scheds(\hat{P})$ because $s \notin scheds(\mathcal{M})$ (since $\mathcal{M}$ and $\mathcal{T}'$ have disjoint sets of output actions by definition of composition of I/O automata), and $s \notin \{\langle a \rangle \mid a \in \Sigma - acts(\mathcal{M})\}$ since s has length > 1. This concludes the proof of our claim. □

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 $\mathcal{M}'$.

Proof. ($\Rightarrow$ *direction*) We assume that we are given a policy $\mathscr{P}$ and a target $\mathcal{T}$ such that $\mathscr{P}$ is soundly enforceable on $\mathcal{T}$ by some monitor $\mathcal{M}$. That is, we assume that 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 $(scheds(\mathsf{hide}_\Phi(\mathcal{M} \times \mathsf{rename}(\mathcal{T}))) | acts(\hat{P})) \subseteq scheds(\hat{P})$.

We have to show that $\mathscr{P}$ is generally soundly enforceable by some monitor $\mathcal{M}'$, or, by definition, that there exists monitor $\mathcal{M}'$ such that for all targets $\mathcal{T}'$ there exists a module $\hat{P}' \in \mathscr{P}$, a renaming function rename', and a hiding function hide' such that $(scheds(\mathsf{hide}'_\Phi(\mathcal{M}' \times \mathsf{rename}'(\mathcal{T}'))) | acts(\hat{P}')) \subseteq scheds(\hat{P}')$.

Let:

1. $\mathcal{M}' = \mathsf{hide}_\Phi(\mathcal{M} \times \mathsf{rename}(\mathcal{T}))$,
2. $\hat{P}' = \hat{P}$,
3. rename' be a function that maps a to a' where $a \in acts(\mathcal{T}'), a' \notin acts(\hat{P})$,
4. $\mathsf{hide}'_\Phi = \mathsf{hide}_\emptyset$.

Now it is easy to see that:
$(scheds(\mathsf{hide}'_\Phi(\mathcal{M}' \times \mathsf{rename}'(\mathcal{T}'))) | acts(\hat{P}')) \subseteq scheds(\hat{P}')$
$\Leftrightarrow$ $(scheds(\mathsf{hide}_\emptyset(\mathsf{hide}_\Phi(\mathcal{M} \times \mathsf{rename}(\mathcal{T})) \times \mathsf{rename}'(\mathcal{T}'))) | acts(\hat{P})) \subseteq scheds(\hat{P})$ (by substitution)
$\Leftrightarrow$ $(scheds(\mathsf{hide}_\Phi(\mathcal{M} \times \mathsf{rename}(\mathcal{T})) \times \mathsf{rename}'(\mathcal{T}')) | acts(\hat{P})) \subseteq scheds(\hat{P})$ (by definition of hiding and the fact that $\Phi = \emptyset$)
$\Leftrightarrow$ $(scheds(\mathsf{hide}_\Phi(\mathcal{M} \times \mathsf{rename}(\mathcal{T}))) | acts\hat{P}) \times (scheds(\mathsf{rename}'(\mathcal{T}')) | acts(\hat{P})) \subseteq scheds(\hat{P})$ (by Theorems 5 and 7 in App. C)
$\Leftrightarrow$ $(scheds(\mathsf{hide}_\Phi(\mathcal{M} \times \mathsf{rename}(\mathcal{T}))) | acts\hat{P}) \times \epsilon \subseteq scheds(\hat{P})$ (by definition of rename' and operator $|$)
$\Leftrightarrow$ $(scheds(\mathsf{hide}_\Phi(\mathcal{M} \times \mathsf{rename}(\mathcal{T}))) | acts\hat{P}) \subseteq scheds(\hat{P})$ (by Theorem 7 in App. C)
Note that the last line is true from our assumption, so we are done.

($\Leftarrow$ *direction*) We assume that we are given a policy $\mathscr{P}$ and a target $\mathcal{T}$. Moreover we assume that $\mathscr{P}$ is generally soundly enforceable by some monitor $\mathcal{M}'$. That is, by definition, we assume that for all targets $\mathcal{T}$ there exists a module $\hat{P} \in \mathscr{P}$, a renaming function rename, and a hiding function hide such that $(scheds(\mathsf{hide}_\Phi(\mathcal{M} \times \mathsf{rename}(\mathcal{T}))) | acts(\hat{P})) \subseteq scheds(\hat{P})$

We have to show that $\mathscr{P}$ is soundly enforceable on $\mathcal{T}$ by some monitor $\mathcal{M}$. That is, we have to show that 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 $(scheds(\mathsf{hide}_\Phi(\mathcal{M} \times \mathsf{rename}(\mathcal{T}))) | acts(\hat{P}')) \subseteq scheds(\hat{P}')$.

This is trivially true, since we can use the module, renaming function, hiding function, and monitor from our assumptions. Since the subset relationship is satisfied for every target, it is also trivially satisfied by $\mathcal{T}$.

□

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:
(**C1**) $acts(\hat{P}) \cap acts(\mathsf{rename}(\mathcal{T}')) \neq \emptyset$, or
(**C2**) $range(\mathsf{rename}) \subseteq acts(\mathcal{M}')$
and $(scheds(\mathcal{M}' \times \mathsf{rename}(\mathcal{T}'))|acts(\hat{P})) \subseteq scheds(\hat{P}))\Big)$.

Proof. To prove the theorem under (**C1**), pick $\mathscr{P} = \{\hat{P}\}$, where $acts(\hat{P}) = \{a\}$, and $scheds(\hat{P}) = \{\epsilon\}$. To prove that there $\exists\mathcal{T}$ and $\exists\mathcal{M}$ such that $\mathscr{P}$ is **specifically soundly enforceable** on $\mathcal{T}$ by $\mathcal{M}$, simply pick any compatible $\mathcal{T}$ and $\mathcal{M}$ such that $scheds(\mathcal{T}) = scheds(\mathcal{M}) = \{\epsilon\}$ (any I/O automata with disjoint start states from the rest of the states will do). Then, by using as renaming function the identity function it is easy to see that $(scheds(\mathcal{M}' \times \mathsf{rename}(\mathcal{T}'))|acts(\hat{P})) \subseteq scheds(\hat{P}))$.

Now we have to show that it is not the case that $\exists\mathcal{M}'$ such that for all targets $\mathcal{T}'$ there exists a module $\hat{P} \in \mathscr{P}$, and a renaming function rename, such that $acts(\hat{P}) \cap acts(\mathsf{rename}(\mathcal{T}')) \neq \emptyset$ and $(scheds(\mathcal{M}' \times \mathsf{rename}(\mathcal{T}'))|acts(\hat{P})) \subseteq scheds(\hat{P}))\Big)$. In other words, we have to show that $\forall\mathcal{M}'$, $\exists\mathcal{T}'$ such that $\forall\hat{P} \in \mathscr{P}$, and $\forall$rename, if $acts(\hat{P}) \cap acts(\mathsf{rename}(\mathcal{T}')) \neq \emptyset$ then $(scheds(\mathcal{M}' \times \mathsf{rename}(\mathcal{T}'))|acts(\hat{P})) \supset scheds(\hat{P}))\Big)$. Pick any $\mathcal{T}'$ which contains only one internal action and exhibits finite sequences of it, i.e., $input(\mathcal{T}') = output(\mathcal{T}') = \emptyset$, $internal(\mathcal{T}') = \{x\}$, and $scheds(\mathcal{T}') = \{a^\star \mid a \in internal(\mathcal{T}')\}$. Then for any renaming function, $scheds(\mathsf{rename}(\mathcal{T}')) \neq \emptyset$. Let any s that belongs to $scheds(\mathsf{rename}(\mathcal{T}'))$, with $|s| > 1$. Then, because the internal actions of the renamed target are disjoint from the monitor's actions (by composition of I/O automata), by Thm. 8, $s \in scheds(\mathcal{M}' \times \mathsf{rename}(\mathcal{T}'))$. But, we assumed that $acts(\hat{P}) \cap acts(\mathsf{rename}(\mathcal{T}')) \neq \emptyset$, i.e., $s|acts(\hat{P}) = s$, and that the only element of $\mathscr{P}$ is $\hat{P}$ which contains only the empty sequence. Thus, $s \notin scheds(\hat{P})$. But because $\epsilon \in scheds(\mathcal{M}' \times \mathsf{rename}(\mathcal{T}'))$, we derive that $(scheds(\mathcal{M}' \times \mathsf{rename}(\mathcal{T}'))|acts(\hat{P})) \supset scheds(\hat{P}))\Big)$.

To prove the theorem under (**C2**), pick $\mathscr{P} = \{\hat{P}\}$, where $acts(\hat{P}) = \Sigma$, $input(\hat{P}) = \emptyset$, and $scheds(\hat{P}) = \{\langle a \rangle \mid a \in acts(\hat{P})\}$, i.e., it contains all sequences of length one. To prove that there $\exists\mathcal{T}$ and $\exists\mathcal{M}$ such that $\mathscr{P}$ is **specifically**

soundly enforceable on $\mathcal{T}$ by $\mathcal{M}$, simply pick $\mathcal{T}$ as the trivial automaton, and $\mathcal{M}$ such that $acts(\mathcal{M}) = \{a\}$, for some $a \in Sigma$, and $scheds(\mathcal{M}) = \{\langle a \rangle \mid a \in acts(\mathcal{M})\}$ (any I/O automaton with two states and one transition suffices). Then, by using as renaming function the identity function it is easy to see that $(scheds(\mathcal{M}' \times \mathsf{rename}(\mathcal{T}'))|acts(\hat{P})) \subseteq scheds(\hat{P}))$.

Now we have to show that it is not the case that $\exists \mathcal{M}'$ such that for all targets $\mathcal{T}'$ there exists a module $\hat{P} \in \mathscr{P}$, and a renaming function rename, such that $range(\mathsf{rename}) \subseteq acts(\mathcal{M}')$ and $(scheds(\mathcal{M}' \times \mathsf{rename}(\mathcal{T}'))|acts(\hat{P})) \subseteq scheds(\hat{P}))\Big)$. In other words, we have to show that $\forall \mathcal{M}', \exists \mathcal{T}'$ such that $\forall \hat{P} \in \mathscr{P}$, and $\forall \mathsf{rename}$, if $range(\mathsf{rename}) \subseteq acts(\mathcal{M}')$ then $(scheds(\mathcal{M}' \times \mathsf{rename}(\mathcal{T}'))$ $|acts(\hat{P})) \supset scheds(\hat{P}))\Big)$. Pick a $\mathcal{T}'$ which contains as internal actions the actions that the monitor does not contain in its signature and exhibits finite sequences of them, i.e., $input(\mathcal{T}') = output(\mathcal{T}') = \emptyset$, $internal(\mathcal{T}') = \Sigma - acts(\mathcal{M})$, and $scheds(\mathcal{T}') = \{a^\star \mid a \in internal(\mathcal{T}')\}$. Then for any renaming function, $scheds(\mathsf{rename}(\mathcal{T}')) \neq \emptyset$. Let any s that belongs to $scheds(\mathsf{rename}(\mathcal{T}'))$, with $|s| > 1$. Then, because the internal actions of the renamed target are disjoint from the monitor's actions (by composition of I/O automata), by Thm. 8, $s \in scheds(\mathcal{M}' \times \mathsf{rename}(\mathcal{T}'))$. But, we assumed that the only element of $\mathscr{P}$ is $\hat{P}$ which contains only one-element sequences. Moreover, since we assumed that $range(\mathsf{rename}) \subseteq acts(\mathcal{M}')$, and $acts(\mathcal{M}') \subseteq acts(\hat{P})$, then $s|acts(\hat{P}) = s$. Thus, $s \notin scheds(\hat{P})$. Also, note that $\mathcal{T}'$ is producing all one-element sequences that the monitor cannot produce, and $\mathcal{M}$ produces all one element sequences that the property requires, since the property does not contain input actions, and thus all the sequences can be under the monitor's control. Thus, the superset relation holds.

□

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 $\mathsf{rename}(\mathcal{T})$.

Proof. We fix a policy $\mathscr{P}$, a module $\hat{P}$, a hiding function $\mathsf{hide}_\Phi()$, and a renaming function $\mathsf{rename}()$, and we assume that there exists a monitor $\mathcal{M}$ such that $\Big(scheds(\mathsf{hide}_\Phi(\mathcal{M} \times \mathsf{rename}(\mathcal{T})))|acts(\hat{P}) \subseteq scheds(\hat{P})\Big)$. We have to show that $\hat{P}$ is input forgiving, prefix closed, and quiescent forgiving for $\mathsf{rename}(\mathcal{T})$.

For the sake of contradiction, assume that $\hat{P}$ is not input forgiving, or not prefix closed, or not quiescent forgiving for $\mathsf{rename}(\mathcal{T})$.

Case: $\hat{P}$ is not input forgiving:

Since $\hat{P}$ is not input forgiving, then either $\epsilon \notin scheds(\hat{P})$ or there exists an $s_1 \in scheds(\hat{P})$, a finite prefix s_2 of s_1, and some sequence of input actions s_3 such that $(s_2; s_3) \notin scheds(\hat{P})$. If we assume the first case of $\epsilon \notin scheds(\hat{P})$ we derive a contradiction since the empty sequence belongs to the schedules of any I/O automaton by definition of executions and schedules of I/O automata. If we

assume the latter case, then we know that $s_2 \in \mathit{scheds}(\mathsf{hide}_\Phi(\mathcal{M} \times \mathsf{rename}(\mathcal{T})))$ by assumption. Let q_n be the state that the monitored target is at after executing the last action of s_2. By definition, every state of an I/O automaton is input enabled. Thus q_n is input enabled, which means that $\forall s' \in (\mathit{input}(\mathsf{hide}_\Phi(\mathcal{M} \times \mathsf{rename}(\mathcal{T}))))^\infty$: $(s_2; s') \in \mathit{scheds}(\mathsf{hide}_\Phi(\mathcal{M} \times \mathsf{rename}(\mathcal{T})))$ (remember we assume no fairness thus it does not matter whether q_n is quiescent or not). But for $s' = s_3$ we get that $(s_2; s_3) \in \mathit{scheds}(\mathsf{hide}_\Phi(\mathcal{M} \times \mathsf{rename}(\mathcal{T})))$ and that $(s_2; s_3) \notin \mathit{scheds}(\hat{P})$ which contradicts our assumption. Thus, in both cases we derived a contradiction, and thus $\hat{P}$ must be input forgiving.

Case: $\hat{P}$ is not prefix closed:

Since $\hat{P}$ is not prefix closed, then there exists some schedule s_1 that belongs to the schedules of $\hat{P}$, but there exists some prefix s_2 of s_1 that does not belong to the schedule of $\hat{P}$, or more formally: $\exists s_1 \in \Sigma^\infty : \big(s_1 \in \mathit{scheds}(\hat{P})\big) \wedge \big(\exists s_2 \in \Sigma^\star : s_2 \preceq s_1 : s_2 \notin \mathit{scheds}(\hat{P})\big)$.

Without loss of generality, assume s_2 is the longest strict prefix of s_1, i.e., it is the longest prefix of s_1 that does not belong to the schedules of $\hat{P}$, and that all prefixes of s_2 belong to the schedules of $\hat{P}$. If $s_2 = a_1, \ldots, a_{n-1}, a_n$ then let $s_2^- = a_1, \ldots, a_{n-1}$ and $s_2^+ = a_1, \ldots, a_{n-1}, a_n, a_{n+1} \preceq s_1$. We know that $s_2^- \in \mathit{scheds}(\hat{P})$ by assumption, $s_2^+ \in \mathit{scheds}(\hat{P})$ because if it was not in the schedules of the property this would be the longest invalid prefix of s_1 which contradicts our choice of s_2, and thus by assumption they both also belong to the schedules of the monitored target $\mathsf{hide}_\Phi(\mathcal{M} \times \mathsf{rename}(\mathcal{T}))$. Let q_n be the state that the monitored target is after executing a_{n-1}, and q_{n+1} be the state before executing a_{n+1}. In order for the automaton to transition from q_n to q_{n+1} it must execute some a_n. But then $s_2^-; a_n = s_2 \in \mathit{scheds}(\mathsf{hide}_\Phi(\mathcal{M} \times \mathsf{rename}(\mathcal{T})))$, while we assumed that $s_2 \notin \mathit{scheds}(\hat{P})$. This contradicts our original assumption, and thus $\hat{P}$ must be prefix closed.

Case: $\hat{P}$ is not quiescent forgiving:

Since $\hat{P}$ is not quiescent forgiving for $\mathsf{rename}(\mathcal{T})$, then there exists some execution $e = q_0, a_1, \ldots, q_n$ of $\mathsf{rename}(\mathcal{T})$ with $q_n \in \mathit{quiescent}(\mathcal{T})$ and $q_i \notin \mathit{quiescent}(\mathcal{T})$ for $0 \leq i < n$ such that either $(\mathit{sched}(e)|\mathit{acts}(\hat{P})) \notin \mathit{scheds}(\hat{P})$ or some prefix t of $(\mathit{sched}(e)|\mathit{acts}(\hat{P}))$ does not belong to the schedules of $\hat{P}$.

By Theorem 7 we know that if $\mathit{sched}(e) \in \mathit{scheds}(\mathsf{rename}(\mathcal{T}))$, then $\mathit{sched}(e) \in \mathit{scheds}(\mathsf{hide}_\Phi(\mathcal{M} \times \mathsf{rename}(\mathcal{T})))$. Thus, $(\mathit{sched}(e)|\mathit{acts}\hat{P}) \in \mathit{scheds}(\mathsf{hide}_\Phi(\mathcal{M} \times \mathsf{rename}(\mathcal{T})))$. But the fact that $(\mathit{sched}(e)|\mathit{acts}(\hat{P})) \notin \mathit{scheds}(\hat{P})$ contradicts our assumption. With the same argument we can show that even if we assume some prefix t of $\mathit{sched}(e)$ we also derive a contradiction. Thus $\hat{P}$ must be quiescent forgiving.

□

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 $\mathcal{M}_2$), if the

policy does not reason about the communication between the monitor and the target[10].

Proof. Take $\mathscr{P} = \{\hat{P}\}$, where $acts(\hat{P}) = output(\hat{P}) = \bigcup_{i \in I} output(\mathcal{T}_i) \cup \bigcup_{i \in I} \mathsf{rename}_{\mathsf{j} \in \mathsf{J}}(output(\mathcal{T}_i))$, $scheds(\hat{P}) = \{\epsilon\} \cup \{\langle a \rangle \mid a \in acts(\hat{P})\}$, and $\hat{P}$ does not reason about the communication between the monitor and the target where I is the set of all targets, and J the set of all renaming functions (note that in the rest of the proof, for purposes of brevity of presentation, we are assuming that the universes of Input, Output, Internal actions are disjoint, and that renaming functions always map actions to fresh actions that are distinct from the Input, Output, and Internal actions of the targets).

For proving the left conjunct of the theorem statement, we have to prove that there exists an input/output-mediating $\mathcal{M}_1$ such that for all targets $\mathcal{T}$ there exists a module $\hat{P} \in \mathscr{P}$, a renaming function rename, such that $(scheds(\mathcal{M}_1 \times \mathsf{rename}(\mathcal{T}))|acts(\hat{P})) = scheds(\hat{P})$.

Let $\mathcal{M}_1$ be the input/output-mediating monitor that has as elements of its signature the following sets: $input(\mathcal{M}_1) = \{\bigcup_{i \in I} input(\mathcal{T}_i)\} \cup \{\bigcup_{i \in I} \mathsf{rename}_{\mathsf{j} \in \mathsf{J}}(output(\mathcal{T}_i))\}$, $output(\mathcal{M}_1) = \{\bigcup_{i \in I} output(\mathcal{T}_i)\} \cup \{\bigcup_{i \in I} \mathsf{rename}_{\mathsf{j} \in \mathsf{J}}(input(\mathcal{T}_i))\}$, $internal(\mathcal{M}_1) = \emptyset$, where I is the set of all targets, and J the set of all renaming functions.

Moreover, let $scheds(\mathcal{M}_1)$ contain no schedules that include more than one output actions from the subset $\{\bigcup_{i \in I} output(\mathcal{T}_i)\}$, i.e., the monitor does not exhibit any output behavior to the environment that contains more than one action. This is easy to do: just exhibit the first valid output action that the target wants to execute, and suppress all future attempts. Now it is easy to see that for all targets $\mathcal{T}$ there exists a module $\hat{P} \in \mathscr{P}$, a renaming function rename, such that $(scheds(\mathcal{M}_1 \times \mathsf{rename}(\mathcal{T}))|acts(\hat{P})) = scheds(\hat{P})$: assume otherwise, i.e., there exists a schedule $s \in scheds(\mathcal{M}_1 \times \mathsf{rename}(\mathcal{T}))$ that is not an element of $scheds(\hat{P})$. Since $scheds(\hat{P})$ contains all possible sequences of length one that contain the output actions of all targets (and all their possible renamings), the only way for $(s \mid acts(\hat{P}))$ not to be an element of the schedules of $\hat{P}$ is to contain output actions and have length larger than 1. However, this is impossible by (1) construction of the monitor, and (2) assumption that the policy does not reason about the communication between the monitor and the target (by Def. 6, all output actions of the target are mediated by the monitor and thus considered part of their communication).

For proving the right conjunct of the theorem statement, we have to prove that it is not the case that there exists an input-mediating $\mathcal{M}_2$ such that for all targets $\mathcal{T}$ there exists a module $\hat{P} \in \mathscr{P}$, a renaming function rename, such that $(scheds(\mathcal{M}_2 \times \mathsf{rename}(\mathcal{T}))|acts(\hat{P})) = scheds(\hat{P})$. In other words, we have to prove that for all input-mediating $\mathcal{M}_2$, there exists a target $\mathcal{T}$, such that for all modules $\hat{P} \in \mathscr{P}$, and for all renaming functions rename: $(scheds(\mathcal{M}_2 \times \mathsf{rename}(\mathcal{T}))|acts(\hat{P})) \neq scheds(\hat{P})$.

[10] If we used trace enforcement, the constraint would be superfluous. We used schedules to remain consistent with other theorems in the paper.

To prove the claim, take any target $\mathcal{T}$ such that $\exists s \in \mathit{scheds}(\mathcal{T})$, and s contains more than two output actions. Let s' be the schedule of the renamed target $\mathsf{rename}(\mathcal{T})$ that corresponds to s. Then, by Thm. 8 s' is contained in $(\mathit{scheds}(\mathcal{M}_2 \times \mathsf{rename}(\mathcal{T}))$ (since the output actions of the target and the monitor are disjoint). Also, s, and thus s' contain more than two output actions. Moreover, by definition of $\hat{P}_i$, $\mathit{acts}(\hat{P}) = \mathit{output}(\hat{P}_i) \supseteq \mathit{output}(\mathsf{rename}(\mathcal{T}))$, for any rename function. And since every element $\hat{P}_i$ of $\mathscr{P}$ does not contain any schedules with more than two output actions, $(s' \mid \mathit{acts}(\hat{P})) \notin \mathit{scheds}(\hat{P})$. This concludes the proof of the claim.

□

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[11],
- (**C2**) $\forall \hat{P} \in \mathscr{P} : \mathit{scheds}(\hat{P})$ is quiescent forgiving for $\mathcal{T}$,
- (**C3**) $\forall \hat{P} \in \mathscr{P} : \mathit{scheds}(\hat{P}) \subseteq \mathit{scheds}(\mathcal{T})$, i.e., the policy does not allow schedeles that cannot be exhibited by the target, and
- (**C4**) $\forall e \in \mathit{execs}(\mathcal{T})$: if e has two quiescent states separated by a sequence of local actions, and any prefix of e ending between those states violates $\mathscr{P}$,
  then all prefixes of e ending between those states violate $\mathscr{P}$ and the two quiescent states are the same.

Proof. ($\Rightarrow$ direction) We assume that we have some arbitrary policy $\mathscr{P}$ and target $\mathcal{T}$, and an input/output-mediating $\mathcal{M}_1$ that specifically precisely enforces $\mathscr{P}$ on $\mathcal{T}$. We need to show that there exists an input-mediating $\mathcal{M}_2$ that specifically precisely enforces $\mathscr{P}$ on $\mathcal{T}$.

Let $\mathcal{M}_2$ be such that (a) its transition relation is the subset of that of $\mathcal{M}_1$'s that deals only with inputs from the environment, i.e., we ignore the part of $\mathcal{M}_1$ that receives input actions from $\mathcal{T}$ and outputs output actions to environment, and (b) for every input i that belongs to the set of inputs that invalidate extensions we simply remove the corresponding transitions: in other words, for every i that invalidates executions, and for every transition of the form $\langle q, i, q' \rangle$, we remove all transitions of the form $\langle q', a, q'' \rangle$, where a is a local action (we keep input actions because of input-enabledness). We will show that if $\mathcal{M}_1$ specifically precisely enforces $\mathscr{P}$ on $\mathcal{T}$ under the above constraints, then $\mathcal{M}_1$ specifically precisely enforces $\mathscr{P}$ on $\mathcal{T}$ also.

First we will show that the part of $\mathcal{M}_1$ that receives inputs from $\mathcal{T}$ and outputs actions to the environment does not do anything "non-trivial" (under the given constraints), i.e., it either outputs nothing or it simply forwards valid actions that $\mathcal{T}$ wants to execute. We do a case analysis on the actions that

[11] As before, if we used trace enforcement, the constraint would be superfluous; thus the constraint is more of a technical issue rather a key idea in the theorem. We used schedules to remain consistent with other theorems in the paper.

$\mathcal{T}$ might execute and prove that the output-mediating part of $\mathcal{M}_1$ is trivial. First, observe that by (**C3**), $\mathcal{M}_1$ cannot arbitrarily add actions that $\mathcal{T}$ might not execute. Second, if $\mathcal{T}$ wants to output some action a that obeys the policy, then because of the precise enforcement constraint, $\mathcal{M}_1$ will have to (eventually) output it. Thus in the case of $\mathcal{M}_1$, a is eventually exhibited by itself, whereas in $\mathcal{M}_2$, a will be exhibited directly by $\mathcal{T}$. Finally,if $\mathcal{T}$ wants to output some action a that disobeys the policy, then a can either be preceded by some input or not. If it is not preceded by some input, then it must be part of quiescent behavior, which by (**C2**) cannot be invalid. So a must be preceded by some input. If it is, then until the next quiescent state all outputs will be invalid by (**C4**), and thus $\mathcal{M}_1$ will not output any of those actions.

Note that the latter is equivalent to not forwarding i to $\mathcal{T}$ since the next quiescent state is the same with the original one, so the target did not make any "progress". This is exactly the construction that corresponds to (b). So under the given constraints the two monitors will both precisely enforce $\mathscr{P}$ on $\mathcal{T}$.

($\Leftarrow$ direction) We assume that we have some arbitrary policy $\mathscr{P}$ and target $\mathcal{T}$, and an input-mediating $\mathcal{M}_2$ that specifically precisely enforces $\mathscr{P}$ on $\mathcal{T}$. We need to construct an input/output-mediating $\mathcal{M}_1$ that specifically precisely enforces $\mathscr{P}$ on $\mathcal{T}$.

We use $\mathcal{M}_2$ to get $\mathcal{M}_1$. Specifically, we use the same transition relation as $\mathcal{M}_2$ which we extend in a manner similar to the construction of a truncation monitor from a truncation automaton (§3), i.e., we add a special state and a queue that buffers the inputs that the $\mathcal{M}_1$ receives from $\mathcal{T}$. Specifically, once $\mathcal{M}_1$ receives some output $\mathsf{rename}(a)$ from $\mathcal{T}$, it records the state it was before starting to receive inputs. If more inputs follow, it adds them to the queue. Once it has finished forwarding to the environment all the outputs that $\mathcal{T}$ wanted to execute (i.e., forward a for each $\mathsf{rename}(a)$ received), it returns to the original state to continue execution (as $\mathcal{M}_2$).

Since $\mathscr{P}$ does not reason about the communication between the monitor and the target, it is easy to see that $(\mathit{scheds}(\mathcal{M}_2 \times \mathsf{rename}(\mathcal{T}))|\mathit{acts}(\hat{P})) = (\mathit{scheds}(\mathcal{M}_1 \times \mathsf{rename}(\mathcal{T}))|\mathit{acts}(\hat{P}))$. Every schedule that $\mathcal{M}_2 \times \mathsf{rename}(\mathcal{T})$ produces is a schedule of $\mathcal{M}_1 \times \mathsf{rename}(\mathcal{T})$, since by construction $\mathcal{M}_2$ does not add any new schedules, and dually, every schedule that $\mathcal{M}_1 \times \mathsf{rename}(\mathcal{T})$ produces is a schedule of $\mathcal{M}_2 \times \mathsf{rename}(\mathcal{T})$, since by construction $\mathcal{M}_2$ does not remove any schedules.

□

Theorem 6. $\forall \mathscr{P} : \forall \mathcal{T}$:
$\mathscr{P}$ 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**) $\forall \hat{P} \in \mathscr{P} : \mathit{scheds}(\hat{P})$ is quiescent forgiving for $\mathcal{T}$,
- (**C3**) $\forall \hat{P} \in \mathscr{P} : \mathit{scheds}(\hat{P}) \subseteq \mathit{scheds}(\mathcal{T})$, and

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

Proof. ($\Rightarrow$ direction) We assume that we have some arbitrary policy $\mathscr{P}$ and target $\mathcal{T}$, and an input/output-mediating $\mathcal{M}_1$ that specifically precisely enforces $\mathscr{P}$ on $\mathcal{T}$. We need to show that there exists an input-mediating $\mathcal{M}_2$ that specifically precisely enforces $\mathscr{P}$ on $\mathcal{T}$.

We construct $\mathcal{M}_2$ by (a) taking the restriction of $\mathcal{M}_1$ that deals only with inputs from the environment, i.e., we ignore the part of $\mathcal{M}_1$ that receives input actions from $\mathcal{T}$ and outputs output actions to environment, and (b) for every input i that belongs to the set of inputs that invalidate extensions we simply remove the corresponding transitions: in other words, for every i that invalidates executions, and for every transition of the form $\langle q, i, q' \rangle$, we remove all transitions of the form $\langle q', a, q'' \rangle$, where a is a local action (we keep input actions because of input-enabledness). We will show that if $\mathcal{M}_1$ specifically precisely enforces $\mathscr{P}$ on $\mathcal{T}$ under the above constraints, then $\mathcal{M}_1$ specifically precisely enforces $\mathscr{P}$ on $\mathcal{T}$ also.

First we will show that the part of $\mathcal{M}_1$ that receives inputs from $\mathcal{T}$ and outputs actions to the environment does not do anything "non-trivial" (under the given constraints), i.e., it either outputs nothing or it simply forwards valid actions that $\mathcal{T}$ wants to execute. We do a case analysis on the actions that $\mathcal{T}$ might execute and prove that the output-mediating part of $\mathcal{M}_1$ is trivial. First, observe that by (**C3**), $\mathcal{M}_1$ cannot arbitrarily add actions that $\mathcal{T}$ might not execute. Second, if $\mathcal{T}$ wants to output some action a that obeys the policy, then because of the precise enforcement constraint, $\mathcal{M}_1$ will have to (eventually) output it. Thus in the case of $\mathcal{M}_1$, a is eventually exhibited by itself, whereas in $\mathcal{M}_2$, a will be exhibited directly by $\mathcal{T}$. Finally,if $\mathcal{T}$ wants to output some action a that disobeys the policy, then a can either be preceded by some input or not. If it is not preceded by some input, then it must be part of quiescent behavior. But since $\mathscr{P}$ is enforceable, then by Thm. 3 it must be quiescent forgiving, i.e., it must be valid – contradiction. So a must be preceded by some input. But, by (**C2**), there is some input i that precedes a after which all extensions are invalid. Thus, i will be the last action appearing on the schedule. This means that since $\mathcal{T}$ can still communicate with $\mathcal{M}_1$, $\mathcal{M}_1$ will suppress all the security relevant behavior following i (i.e., it will trivially output nothing).

Note that the latter is equivalent to not forwarding i to $\mathcal{T}$ (or any future inputs) and just continue execution by receiving inputs from the environment. This is exactly the construction that corresponds to (b). So under the given constraints the two monitors will both precisely enforce $\mathscr{P}$ on $\mathcal{T}$.
($\Leftarrow$ direction)

This is the same as the $\Leftarrow$ Direction in the proof of Thm. 5.

□

Lemma 1. Given a schedule module $\hat{P}$ and a target $\mathcal{T}$, then there exists a truncation monitor $\mathcal{M}_T$ such that $\hat{P}$ is specifically precisely enforceable on $\mathcal{T}$ by $\mathcal{M}_T$ if:

(**C1**) $\forall \hat{P} \in \mathscr{P} : scheds(\hat{P}) \subseteq scheds(\mathcal{T})$,

(**C2**) $\forall t \in scheds(\mathcal{T}) :$
$t \notin scheds(\hat{P}) \Rightarrow \exists s \preceq t :$
$\forall k \in (acts(\mathcal{T}) \cup acts(\hat{P}))^{\omega} : s \prec k :$
$k \in scheds(\mathcal{T}) \Rightarrow k \notin scheds(\hat{P})$, and

(**C3**) $\hat{P}$ does not reason about the communication between the target and the monitor and the internal actions of the target.

Proof. We will assume that the target can be terminated by a *stop* action. We use a construction similar to the one that was described in §3. Let $\mathcal{T}' = f(\mathcal{T})$, where f is a bijective renaming function that α-renames $\mathcal{T}$.

1. $sig(M_T) = \langle input(M_T), internal(M_T), output(M_T) \rangle$, where:
   (i) $input(M_T) = f(output(\mathcal{T}))$,
   (ii) $internal(M_T) = \emptyset$,
   (iii) $output(M_T) = output(T) \cup \{stop\}$.
2. $states(M_T) = ((input(M_T))^{\infty}) \cup \{halt\}$,
3. $start(M_T) = \{\epsilon\}$,
4. $trans(M_T) =$
   $\{\langle \sigma, \iota, \sigma; \iota \rangle \mid \langle \sigma \rangle \in states(M_T) \text{ and } \iota \in input(M_T)\}$
   $\cup \{\langle \alpha; \sigma, f^{-1}(\alpha), \sigma \rangle \mid (\alpha; \sigma) \in states(M_T) \text{ and } (\alpha; \sigma) \in scheds(\hat{P})\}$
   $\cup \{\langle \alpha; \sigma, stop, halt \rangle \mid (\alpha; \sigma) \in states(M_T) \text{ and } (\alpha; \sigma) \notin scheds(\hat{P})\}$
   $\cup \{halt, \iota, halt \mid \iota \in input(M_T)\}$,
5. Each action in $local(M_T)$ defines a unique equivalence class.

It is easy to see that $scheds(M_T \times f(\mathcal{T}))|acts(\hat{P}) = scheds(\hat{P})$. First, note that by construction M_T never outputs any actions that $\mathcal{T}$ does not want to execute: by (**C1**) this guarantees that there is no schedule in the module that the target can output and the monitor does not. Moreover, by (**C3**), even if the target executes some invalid internal action, or attempts to execute an invalid output action, this is opaque to the module. Finally, with a simple inductive argument over the states of M_T we can prove that M_T maintains the invariant that every schedule that the target wants to execute is eventually output only if it is valid. And this is exactly what (**C2**) describes. Thus, the monitor outputs every schedule that the property reasons about.

□

Theorem 7. $\forall \mathscr{P}$: $\forall \mathcal{T}$: $\mathscr{P}$ is specifically precisely enforceable on $\mathcal{T}$ by some truncation monitor $\mathcal{M}_1$ iff $\mathscr{P}$ is specifically precisely enforceable on $\mathcal{T}$ by some edit monitor $\mathcal{M}_2$, if: $\forall \hat{P} \in \mathscr{P}$:

(**C1**) $\forall \hat{P} \in \mathscr{P} : scheds(\hat{P}) \subseteq scheds(\mathcal{T})$,

(**C3**) $\forall \hat{P} \in \mathscr{P}$: $\forall t \in scheds(\mathcal{T}) :$
$t \notin scheds(\hat{P}) \Rightarrow \exists s \preceq t :$
$\forall k \in (acts(\mathcal{T}) \cup acts(\hat{P}))^{\omega} : s \prec k :$

$k \in \mathit{scheds}(\mathcal{T}) \Rightarrow k \notin \mathit{scheds}(\hat{P})$, and

(**C3**) $\hat{P}$ does not reason about the communication between the target and the monitor and the internal actions of the target.

Proof. ($\Rightarrow$ direction) This direction is easy. We have a truncation monitor $\mathcal{M}_1$ that specifically precisely enforces $\mathscr{P}$ on $\mathcal{T}$, and we want to build an edit monitor $\mathcal{M}_2$ that specifically precisely enforces $\mathscr{P}$ on $\mathcal{T}$. The construction is the same as described in [19]: the halting behavior of $\mathcal{M}_1$ is modeled in $\mathcal{M}_2$ by suppressing all future actions that $\mathcal{T}$ would execute. Thus, $\mathcal{M}_2$ will produce exactly the schedules that $\mathcal{M}_1$ will produce. The only difference is that in the case of the edit monitor $\mathcal{T}$ is not halted and might continue executing actions. But, by constraint (**C3**), all internal and output actions of the target are opaque to the policy, so the two monitored systems will produce exactly the same schedules.

($\Leftarrow$ direction) For this direction we assume that we have an edit monitor $\mathcal{M}_2$ that specifically precisely enforces $\mathscr{P}$ on $\mathcal{T}$, and we want to show that there exists a truncation monitor $\mathcal{M}_1$ that specifically precisely enforces $\mathscr{P}$ on $\mathcal{T}$. Let $\hat{P}$ be the element of $\mathscr{P}$ for which $\mathcal{M}_2$ specifically precisely enforces $\mathscr{P}$ on $\mathcal{T}$. First, we note that by constraint (**C1**), $\mathcal{M}_2$ will not insert any new actions (and schedules) that the target would not exhibit. Also, by (**C2**), any schedule of the target that is invalid, becomes invalid at some finite (discrete) point of the schedule, and remains invalid for the rest of the schedule. This is exactly the definition of safety [19], applied to $\mathcal{T}$ (instead of an arbitrary action-universe). Thus, we know that a truncation monitor $\mathcal{M}_1$ that enforces $\hat{P}$ exists by Lemma 1. Note that the constraints of Lemma 1 are satisfied because of the assumptions of this theorem. Thus, $\mathcal{M}_1$ specifically precisely enforces $\mathscr{P}$ on $\mathcal{T}$.

□