Journal of Computer Security 0 (0) 1
IOS Press

Probabilistic Cost Enforcement of Security Policies

Yannis Mallios [a,*], Lujo Bauer [b] Dilsun Kaynar [c] Fabio Martinelli [d] and Charles Morisset [e]
[a] *Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, USA*
E-mail: mallios@cmu.edu
[b] *Department of Electrical and Computer Engineering, and Institute for Software Research, Carnegie Mellon University, Pittsburgh, PA, USA*
E-mail: lbauer@cmu.edu
[c] *Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA*
E-mail: dilsun@cs.cmu.edu
[d] *Istituto di Informatica e Telematica, National Research Council, Pisa, Italy*
E-mail: Fabio.Martinelli@iit.cnr.it
[e] *Newcastle University, Newcastle, UK*
E-mail: charles.morisset@newcastle.ac.uk

Abstract. This paper presents a formal framework for run-time enforcement mechanisms, or monitors, based on probabilistic input/output automata [9,10], which allows for the modeling of complex and interactive systems. We associate with each trace of a monitored system (i.e., a monitor interposed between a system and an environment) a probability and a real number that represents the cost that the actions appearing on the trace incur on the monitored system. This allows us to calculate the probabilistic (expected) cost of the monitor and the monitored system, which we use to classify monitors, not only in the typical sense, e.g., as sound and transparent [25], but also at a more fine-grained level, e.g., as cost-optimal or cost-efficient. We show how a cost-optimal monitor can be built using information about cost and the probabilistic future behavior of the system and the environment, showing how deeper knowledge of a system can lead to construction of more efficient security mechanisms.

Keywords: Security Policies, Monitoring, Cost, Probabilistic I/O automata, Policy Enforcement

1. Introduction

A common approach to enforcing security policies on untrusted software is run-time monitoring. Run-time monitors, e.g., firewalls and intrusion detection systems, observe the execution of untrusted applications or systems, e.g., web browsers and operating systems, and ensure that their behavior adheres to a security policy.

*Corresponding author. E-mail: mallios@cmu.edu.

0926-227X/0-1900/$27.50 © 0 – IOS Press and the authors. All rights reserved

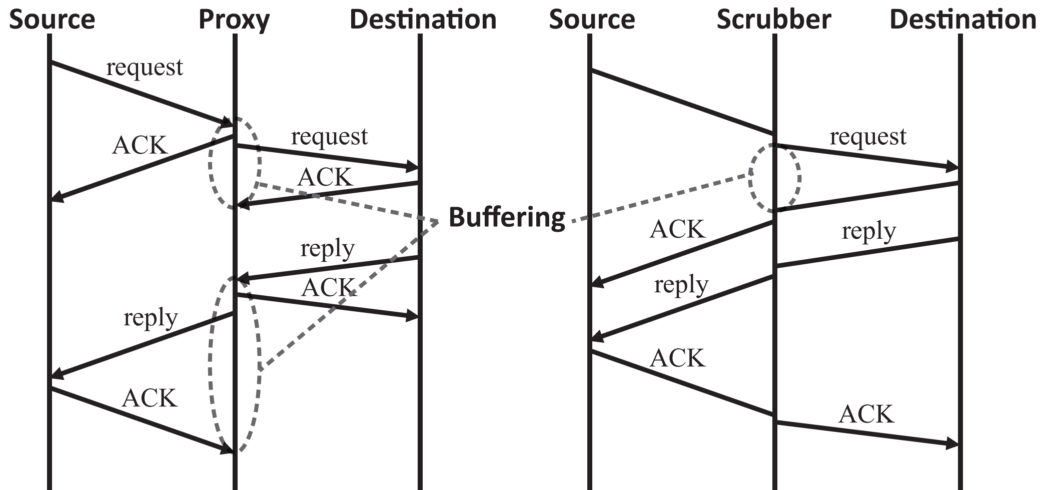


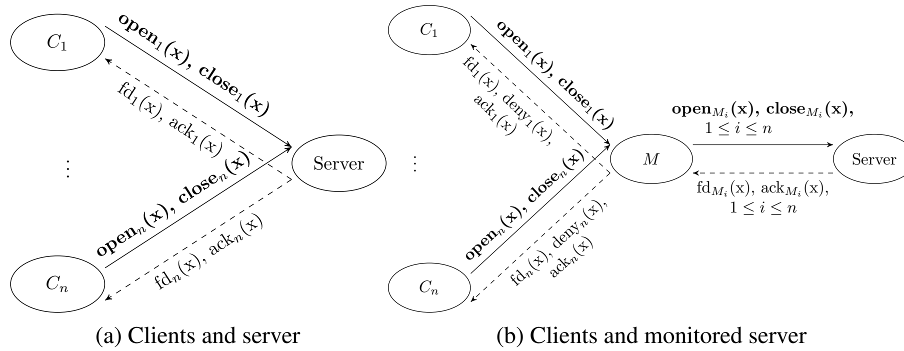
Fig. 1. TCP transport layer proxies and scrubbers. The circled portions represent the amount of time that data is buffered.

Given the ubiquity of run-time monitors and the negative impact they have on the overall security of the system if they fail to operate correctly, it is important to have a good understanding of their behavior and strong guarantees about their correctness and efficiency. Such guarantees can be achieved through the use of formal reasoning.

Schneider introduced security automata [36], an automata-based framework to formally model and reason about run-time enforcement of security policies. Several extensions have been proposed to investigate different definitions of and requirements for enforcement, such as soundness, transparency, and effectiveness (e.g., [25]). A common observation is that once requirements for enforcement are set, more than one implementation of a monitor might be able to fulfill them.

Two examples of common run-time enforcement mechanisms are transport layer proxies and TCP scrubbers [27]. Both of these convert ambiguous TCP flows to unambiguous ones, thereby preventing attacks that seek to avoid detection by network intrusion detection systems (NIDS). Transport layer proxies interpose between a client and a server and create two connections: one between the client and the proxy, and one between the proxy and the server. TCP scrubbers leave the bulk of the TCP processing to the end points: they maintain the current state of the connection and a copy of packets sent by the external host but not acknowledged by the internal receiver. Fig. 1 (adapted from [27]) depicts the differences between the two mechanisms in a specific scenario. Although both mechanisms can correctly enforce the same high-level “no ambiguity” policy, the proxy requires twice the amount of buffering as the scrubber, which suggests that the proxy is more costly (in terms of computational resources).

Recent work has started looking at cost as a metric to classify and compare such monitors. Drabik et al. introduced a framework that calculated the overall cost of enforcement based on costs assigned to the enforcement actions performed by the monitor [16]; this framework can be used to calculate and compare the cost of different monitors’ implementations. This framework provides means to reason about cost-aware enforcement, but its enforcement model does not capture interactions between the target and its environment, including the monitor; recent work has shown that capturing such interactions can be valuable [29]. In addition, in practice the cost of running an application may depend on the ordering of its actions, which may in turn depend on the scheduling strategy.



(a) Clients and server (b) Clients and monitored server

Fig. 2. Diagrams of interposing a Monitor between Clients and Server

In practice, when administrators evaluate the cost of enforcing a security policy, they typical take into account, besides costs, the likelihood of attacks or other specific desirable or undesirable events, i.e., a probabilistic model of the system's behavior. For example, a security policy that describes how to protect a system against different attacks might depend on the probability that these attacks, e.g., a DDOS attack or insider attack, will occur against that particular system. With these probabilities and an estimation of the cost of the impact of the attacks, administrators can calculate the expected costs of the attacks, and then estimate the optimal cost of a monitor to defend against the attacks. For instance, they might decide to invest more resources on a mechanism to defend against DDOS attacks, and fewer resources to defend against insider attacks, because for their systems the former has a higher expected cost.

Running example As another example (which we will use as a running example in this paper), consider the scenario illustrated in Fig. 2a. A file server S is hosting a password file with high integrity requirements. Clients (C_1 through C_n in the figure) request to open or close the file and the server responds to the requests by returning a file descriptor or an acknowledgment that the file was closed successfully. Although not shown in the figure, assume that each client has a number of users who need to get authenticated to use some of the client's services. In order for the users to get authenticated, each client needs to access the file on S to verify the users' passwords. Finally, each client has an Service Level Agreement (SLA) with the server that states that each time the client is denied access, a fixed amount is paid by the server to the client. In this example, the goal of the server is to minimize the number of clients that do not get access to the file, in order to minimize the amount of money it has to pay due to the SLA.

Although not depicted in the figure, let us also assume that the users can update their passwords to S. The concurrent access to the file by multiple clients may compromise the integrity of the file. This compromise can incur a high cost to the server (e.g., penalty costs by the SLA, since the clients will not be able to access the file, costs for recovering the file, etc.). In such cases (where multiple concurrent accesses to a file could compromise its integrity), it is typical to require that access to the file is "locked" to at most one client (and user) at a time. This requirement can be seen as a security policy P stating that at most one client at a time can access a particular file. One way to enforce P is to interpose a monitor between the clients and the server, as shown in Fig. 2b. The monitor has the ability to *deny* access to a file requested by a client if the file is already in use. Since only one client can access the file at a time, and may use it for an arbitrary amount of time, the most cost-efficient solution for the server is to give priority to clients that make many requests and use the file for the least amount of time. This way, there are fewer denied accesses, and the server has to pay a smaller amount of money.

A monitor can use past usage data to identify clients that make fewer requests and use the file for a shorter amount of time (e.g., because they have fewer users, or fewer services) . Through this data the monitor has a probabilistic estimate of the likelihood that a client will request to access the file. Given these estimates, the monitor can optimize its strategy to minimize the expected cost for the server by, for example, giving priority to clients that are more likely to make multiple requests withing a given time frame. Thus, by combining probabilistic knowledge with cost metrics, we are able to reason about, and compare, monitor designs in a way that is closer to how monitors are selected in practice.

Contributions The main contribution of this paper is a formal framework that enables us to (1) model monitors that interact with probabilistic targets and environments (i.e., targets and environments whose behavior we can characterize probabilistically), (2) check whether such monitors enforce a given security policy, and (3) calculate and compare their cost of enforcement. More precisely:

1. Our framework is based on probabilistic I/O automata [9,10]. This allows us to reason about partially ordered events in distributed and concurrent systems, and the probabilities of events and sequences of events.
2. We extend probabilistic I/O automata with *abstract schedulers* to allow fair comparison of systems where a policy is enforced on a target by different monitors.
3. We define cost security policies and cost enforcement, richer notions of (boolean) security policies and enforcement [36]. Cost security policies assign a cost to each trace, allowing richer classification of traces than just as bad or good. We also show how to encode boolean security policies as cost security policies.
4. Finally, we show how to use our framework to compare monitors' implementations and we identify the sufficient conditions for constructing cost-optimal monitors.

2. Overview and Practical Considerations

In this section we briefly discuss the outline of this paper, the problems that we focus on, how we use our framework to solve them, and, finally, some practical considerations.

In §3 we briefly review probabilistic I/O automata (PIOA) [9,10], which we build on in this paper. PIOA allow us to model monitors (as automata) and probabilistic knowledge we have about their environment (as probabilities over transitions). In §3.4 we show how to use PIOA to model our running example of the clients, server, and monitor, and how to calculate probabilities of their executions. Typically these calculations rely on probabilistic knowledge about the environment of a monitor.

Obtaining accurate information about the probability of future events is often very difficult. Thus, in practice, these probabilities must be estimated. Sometimes this is accomplished by logging a system's past interactions with the environment (or obtaining such logs for similar systems) and using this data to infer the probabilities for future events [37]. New, or small, companies might not have access to such data; in this case probabilities can be assigned based on qualitative criteria (e.g., the security manager assigns probabilities based on her experience and knowledge, or uses techniques such as threat categorization [37], or probability ranges rather than point estimates [8]). In this paper we assume that such probabilistic knowledge is available.

In §4 we show how to extend PIOA by adding costs to the model. We model costs using the same mathematical representations as probabilities (i.e., measure theory and cones of traces). The addition of costs to PIOA allows us to calculate the expected costs of monitors. These calculations are very similar

to the calculations of probabilities described in §3.4. We also discuss how one can calculate costs of monitors based on costs of individual events (or actions). This is relevant in practice, since we typically have information about the cost of individual events, from which we can infer or calculate long term costs. For example, we can calculate (or approximate) the running time of a program from knowing how many CPU cycles each individual instruction requires. Similarly to probabilities, calculating the costs of individual events (or executions) depends highly on the specifics of the scenarios we are modeling. For example, the exact cost incurred by a corrupted password file might depend on the existence of a backup, the cost of recovery, the cost of downtime because users cannot access resources, etc. Obtaining and deriving costs of operations bears similar difficulties to the ones we described above for obtaining accurate information on probabilities of future events. Since the goal of this paper is not to examine how such costs can be derived accurately, but rather how they can be used in the context of comparing monitoring designs, we will assume that such costs have already been derived.

As we discussed in §1, one of the goals of this paper is to allow us to choose which of two monitors—both of which may correctly enforce a security policy—minimizes or maximizes some cost of interest. For instance, in our running example, if we have two monitors that correctly enforce the given security policy, we may want to know which one minimizes the average number of clients who are denied access to the file (because some other client has already gained access). Before discussing how to compare two monitors, we need to define what it means for a monitor to enforce a security policy. This is the goal of §5. Previous work has focused on policies that classify executions of a system as either valid or invalid [36]. Since one of our goals in this paper is to compare monitors, we extend the definition of a security policy to a more granular one that classifies monitors (and their executions) according to their expected costs.

Finally, in §6 we show how monitors that have been modeled in our framework can be compared in terms of expected costs. To compare monitors we introduce some technical machinery, such as abstract schedulers (a contribution of this paper). These tools are necessary to overcome difficulties that often arise in frameworks that deal with interaction, probabilities, and costs. We show using our framework that under certain constraints, a monitor can be the optimal one for enforcing a particular security policy. We also discuss sufficient conditions for the construction of such optimal monitors. Our goal is to emphasize both the existence of optimal monitors, and that, in the general setting, there might be no automated way or algorithm to construct optimal monitors (unless the scenarios under consideration are sufficiently constrained). In practice, this means that we cannot have a generic algorithm that will take as input a security policy, probabilistic knowledge about the environment, and cost estimates of attacks and defenses, and will output a monitor that minimizes (or maximizes) the expected cost for enforcing the given policy. However, the constraints we identify using our framework apply to many practical applications, which means that there are many real-world scenarios in which we can construct (verifiably) optimal monitors.

3. Background

We introduce our notation in §3.1 and then in §3.2 briefly review probabilistic I/O automata (PIOA) [9, 10], which we build on in this paper: more details can be found in standard PIOA references, e.g., [9,10]. In §3.3 we extend PIOA by introducing the notion of *abstract schedulers*, which we use in the cost comparison of monitors in §6. Finally, in §3.4 we show how to use PIOA to model practical scenarios through a running example that we will use in the rest of the paper to illustrate the main ideas of our framework.

3.1. Preliminaries

We write $\mathbb{R}^{\geq 0}$ and $\mathbb{R}^{+}$ for the sets of nonnegative real numbers and positive real numbers respectively. Given a function $f : X \to Y$, we write $\mathsf{dom}(f)$ for the domain of f, i.e, X, and $\mathsf{range}(f)$ for the range of f, i.e., Y. Given some $X' \subseteq X$, we write $f \restriction X'$ for the function whose domain is X' and range is $\{f(x) \mid x \in X'\}$.

A $\sigma-$field over a set X is a set $\mathcal{F} \subseteq 2^X$ that contains the empty set and is closed under complement and countable union. A pair $(X, \mathcal{F})$ where $\mathcal{F}$ is a $\sigma-$field over X, is called a *measurable space*. A measure on a measurable space $(X, \mathcal{F})$ is a function $\mu : \mathcal{F} \to [0, \infty]$ that is countably additive: for each countable family $\{X_i\}_i$ of pairwise disjoint elements of $\mathcal{F}$, $\mu(\cup_i X_i) = \Sigma_i \mu(X_i)$.

A *probability measure* on $(X, \mathcal{F})$ is a measure on $(X, \mathcal{F})$ such that $\mu(X) = 1$. A *sub-probability measure* on $(X, \mathcal{F})$ is a measure on $(X, \mathcal{F})$ such that $\mu(X) \leq 1$. A *discrete probability measure* on a set X is a probability measure μ on $(X, 2^X)$, such that, for each $C \subseteq X$, $\mu(C) = \sum_{c \in C} \mu(\{c\})$. A *discrete sub-probability measure* on a set X is a sub-probability measure μ on $(X, 2^X)$, such that, for each $C \subseteq X$, $\mu(C) = \sum_{c \in C} \mu(\{c\})$. We define $\mathsf{Disc}(X)$ and $\mathsf{SubDisc}(X)$ to be, respectively, the set of discrete probability measures and discrete sub-probability measures on X. In the sequel, we often omit the set notation when we refer to the measure of a singleton set.

A *support* of a probability measure μ is a measurable set C such that $\mu(C) = 1$. If μ is a discrete probability measure, then we denote by $\mathsf{supp}(\mu)$ the set of elements that have non-zero measure (thus $\mathsf{supp}(\mu)$ is a support of μ). We let $\delta(x)$ denote the *Dirac measure* for x, the discrete probability measure that assigns probability 1 to $\{x\}$.

A *signed measure* on $(X, \mathcal{F})$ is a function $\nu : \mathcal{F} \to [-\infty, \infty]$ such that: (1) $\nu(\emptyset) = 0$, (2) ν assumes at most one of the values $\pm\infty$, and (3) for each countable family $\{X_i\}_i$ of pairwise disjoint elements of $\mathcal{F}$, $\nu(\cup_i X_i) = \Sigma_i \mu(X_i)$ with the sum converging absolutely if $\nu(\cup_i X_i)$ is finite.

Given two discrete measures μ_1, μ_2 on $(X, 2^X)$ and $(Y, 2^Y)$, respectively, we denote by $\mu_1 \times \mu_2$ the *product measure*, that is, the measure on $(X \times Y, 2^{X \times Y})$ such that $\mu_1 \times \mu_2(x, y) = \mu_1(x) \cdot \mu_2(y)$ (i.e., component-wise multiplication) for each $x \in X, y \in Y$.

If $\{\rho_i\}_{i \in I}$ is a countable family of measures on $(X, \mathcal{F}_X)$ and $\{\rho_i\}_{i \in I}$ is a family of non-negative values, then the expression $\Sigma_{i \in I} p_i \rho_i$ denotes a measure ρ on $(X, \mathcal{F}_X)$ such that for each $C \in \mathcal{F}_X$, $\rho(C) = \Sigma_{i \in I} p_i \cdot \rho_i(C)$.

A function $f : X \to Y$ is said to be measurable from $(X, \mathcal{F}_X) \to (Y, \mathcal{F}_Y)$ if the inverse image of each element of $\mathcal{F}_Y$ is an element of $\mathcal{F}_X$; that is, for each $C \in \mathcal{F}_Y$, $f^{-1}(C) \in \mathcal{F}_X$. Note that, if $\mathcal{F}_X$ is 2^X, then any function $f : X \to Y$ is measurable $(X, \mathcal{F}_X) \to (Y, \mathcal{F}_Y)$ for any $(Y, \mathcal{F}_Y)$. Given measurable f from $(X, \mathcal{F}_X) \to (Y, \mathcal{F}_Y)$ and a measure μ on $(X, \mathcal{F}_X)$, the function $f(\mu)$ defined on $\mathcal{F}_Y$ by $f(\mu)(C) = \mu(f^{-1}(C))$ for each $C \in Y$ is a measure on $(Y, \mathcal{F}_Y)$ and is called the *image measure* of μ under f. If $\mathcal{F}_X = 2^X$, $\mathcal{F}_Y = 2^Y$, and μ is a sub-probability measure, then the image measure $f(\mu)$ is a sub-probability satisfying $f(\mu)(Y) = \mu(X)$.

3.2. Probabilistic I/O Automata

An *action signature* S is a triple of three disjoint sets of actions: *input*, *output*, and *internal* actions (denoted as $input(S)$, $output(S)$, and $internal(S)$). The *external* actions $extern(S)$=$input(S) \cup output(S)$ model the interaction of the automaton with the evironment. Given a signature S we write $acts(S)$ for the set of all actions contained in the signature, i.e., $acts(S) = input(S) \cup output(S) \cup internal(S)$.

A probabilistic I/O automaton (PIOA) P is a tuple $(sig(P), Q(P), \bar{q}_P, R(P))$, where: (1) $sig(P)$ is an action signature; (2) $Q(P)$ is a (possibly infinite) set of *states*; (3) $\bar{q}_P$ is a *start state*, with $\bar{q}_P \in Q(P)$; and (4) $R(P) \subseteq Q(P) \times acts(P) \times \mathsf{Disc}(Q(P))$ is a transition relation, where $\mathsf{Disc}(Q(P))$ is the set of discrete probability measures on $Q(P)$.

Given a PIOA P, we write $acts(P)$ for $acts(sig(P))$. We assume that P satisfies the following conditions: (i) *Input enabling:* For every state $q \in Q(P)$ and input action $\alpha \in input(P)$, α is enabled[1] in q; and (ii) *Transition determinism:* For every state $q \in Q(P)$ and action $\alpha \in acts(P)$, there is at most one $\mu \in \mathsf{Disc}(Q(P))$ such that $(q, \alpha, \mu) \in R(P)$. If there exists exactly one such μ, it is denoted by $\mu_{q,\alpha}$, and we write $\mathsf{tran}_{q,\alpha}$ for the transition $(q, \alpha, \mu_{q,\alpha})$.

A *non-probabilistic execution* e of P 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: (1) $q_0 = \bar{q}_P$, and (2) for every non-final i, there is a transition $(q_i, a_{i+1}, \mu) \in R(P)$ with $q_{i+1} \in \mathsf{supp}(\mu)$.

We write $\mathsf{fstate}(e)$ for q_0, and, if e is finite, we write $\mathsf{lstate}(e)$ for the last state of e. The *trace* of an execution e, written $trace(e)$, is the restriction of e to the set of external actions of P. We say that t is a *trace* of P if there is an execution e of P such that $trace(e) = t$. We write $\mathsf{laction}(t)$ for the last action of t, when t is finite. We use $execs(P)$ and $traces(P)$ (resp., $execs^*(P)$ and $traces^*(P)$) to denote the set of all (resp., all finite) executions and traces of an PIO automaton P.

The symbol λ denotes the empty sequence. We write $e_1; e_2$ for the concatenation of two executions the first of which has finite length and $\mathsf{lstate}(e_1) = \mathsf{fstate}(e_2)$. When σ_1 is a finite prefix of σ_2, we write $\sigma_1 \preceq \sigma_2$, and, if a strict finite prefix, $\sigma_1 \prec \sigma_2$. Given a finite set A, $A^\star$ denotes the set of finite sequences of elements of A and A^ω the set of infinite sequences of elements of A. The set of all finite and infinite sequences of elements of A is $A^\infty = A^\star \cup A^\omega$. Thus, if A is a set of actions Σ, then $\Sigma^\star$ denotes the set of finite sequences of actions and Σ^ω the set of infinite sequences of actions. The set of all finite and infinite sequences of actions is $\Sigma^\infty = \Sigma^\star \cup \Sigma^\omega$.

An automaton that models a complex system can be constructed by *composing* automata that model the system's components. When composing automata P_i, where $i \in I$ and I is finite 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 $P_i : i \in I$ or called compatible if for all $i, j \in I$: (1) $output(P_i) \cap output(P_j) = \emptyset$; (ii) $internal(P_i) \cap acts(P_j) = \emptyset$. When the signatures are compatible we say that the corresponding automata and modules are compatible too. The composition $P = \prod_{i \in I} P_i$ of a set of compatible automata $\{P_i : i \in I\}$ is defined as:

1. $sig(P) = \prod_{i \in I} sig(P_i) = \Big(output(P) = \cup_{i \in I} output(P_i), \quad internal(P) = \cup_{i \in I} internal(P_i), input(P) = \cup_{i \in I} input(P_i) - \cup_{j \in I} output(P_j)\Big)$;
2. $Q(P) = \prod_{i \in I} Q(P_i)$;
3. $\bar{q}_P = \prod_{i \in I} \bar{q}_{P_i}$;
4. $R(P)$ is equal to the set of triples $(q, a, \prod_{i \in I} \mu_i)$ such that:
   (a) a is enabled in some $q_i \in q$, and
   (b) for all $i \in I$ if $a \in acts(P_i)$ then $(q_i, a, \mu_i) \in R(P_i)$, otherwise $\mu_i = \delta(q_i)$.

[1]If a PIOA P has a transition $(q, \alpha, \mu) \in R(P)$ then we say that action α is *enabled* in state q.

Schedulers. Nondeterministic choices in P are resolved using a *scheduler*. A *scheduler* for P is a function $\sigma : execs^*(P) \rightarrow \mathsf{SubDisc}(R(P))$ s.t., if $(q, a, \mu) \in \mathsf{supp}(\sigma(e))$ then $q = \mathsf{lstate}(e)$. Thus, σ decides (probabilistically) which transition (if any) to take after each finite execution e. Since this decision is a discrete sub-probability measure, it may be the case that σ chooses to *halt* after e with non-zero probability: $1 - \sigma(e)(R(P)) > 0$.

A scheduler σ together with a finite execution e *generates* a measure $\epsilon_{\sigma,e}$ on the $\sigma-$field $\mathcal{F}_P$ generated by cones of executions, where the cone $C_{e'}$ of a finite execution e' is the set of executions that have e' as prefix. The construction of the $\sigma-$field is standard [9]. The measure of a cone $\epsilon_{\sigma,e}(C_{e'})$ is defined recursively as:

1. 0, if $e' \not\preceq e$ and $e \not\preceq e'$;
2. 1, if $e' \preceq e$;
3. $\epsilon_{\sigma,e}(C_{e''})\mu_{\sigma(e'')}(a, q)$, if e' is of the form $e''\ a\ q$, $e \preceq e''$. Here, $\mu_{\sigma(e'')}(a, q)$ is defined to be $\sigma(e'')(\mathsf{tran}_{\mathsf{lstate}(e''),a})\mu_{\mathsf{lstate}(e''),a}(q)$, that is, the probability that $\sigma(e'')$ chooses a transition labeled by a and that the new state is q.

Standard measure theoretic arguments ensure that $\epsilon_{\sigma,e}$ is a probability measure. For full details the reader is referred to [9]. The measure of a cone of an execution e corresponds, intuitively, to the probability for e to happen.

We note that the *trace* function is a measurable function from $\mathcal{F}_P$ to the $\sigma-$field $\mathcal{F}_{P_T}$ generated by cones of traces. Thus, given a probability measure ϵ on $\mathcal{F}_P$, we define the *trace distribution* of ϵ, denoted $\mathsf{tdist}(\epsilon)$ to be the image measure of ϵ under *trace*, i.e., for each cone of traces C_t, $\mathsf{trace}(\epsilon)(C_t)$ $= \epsilon(\mathsf{trace}^{-1}(C_t))$. We denote by $\mathsf{tdists}(P)$ the set of trace distributions of (probabilistic executions of) P.

3.3. Abstract Schedulers

In this section we introduce abstract schedulers, a novel extension of PIOA and one of the contributions of this paper. Abstract schedulers are used in the cost comparison of monitors (§6). Schedulers, as defined above, assign probabilities to specific PIOA and their transitions. Thus, one scheduler cannot be defined for two different monitors. As we discuss in §6, there are certain difficulties when trying to fix a scheduler for two different monitors even if they are defined over the same signature. We overcome these difficulties, by introducing abstract schedulers that are defined for all possible traces over a given signature. These abstract schedulers can then be refined to schedulers for particular PIOA.

Given a signature S, an *abstract scheduler* τ for S is a function $\tau : (extern(S))^* \rightarrow \mathsf{SubDisc}(extern(S))$. τ decides (probabilistically) which action appears after each finite trace[2] t. Note that an abstract scheduler τ assigns probabilities to all possible (finite) traces over the given signature.

An abstract scheduler τ together with a finite trace t *generate* a measure $\zeta_{\tau,t}$ on the $\sigma-$field $\mathcal{F}_{P_T}$ generated by cones of traces, where the cone $C_{t'}$ of a finite trace t' is the set of traces that have t' as prefix. The measure of a cone $\zeta_{\tau,t}(C_{t'})$ is defined recursively as:

1. 0, if $t' \not\preceq t$ and $t \not\preceq t'$;
2. 1, if $t' \preceq t$;
3. $\zeta_{\tau,t}(C_{t''})\tau(t'')(\{a\})$, if t' is of the form $t''; a$, $t \preceq t''$.

---

[2]Note that the term “trace” is overloaded: it refers to either the result of applying the function trace to an execution e or to a sequence of external actions. It will be clear from the context to which of the two cases we refer each time.

Standard measure theoretic arguments ensure that $\zeta_{\tau,t}$ is well defined and a probability measure.

Refining abstract schedulers. Abstract schedulers give us (sub-)probabilities for all possible traces over a given signature. However, a given PIOA P might exhibit only a subset of all those possible traces. Thus, we would like to have a way to *refine* an abstract scheduler τ to a scheduler σ that corresponds to the particular PIOA P and is "similar" to τ w.r.t. assigning probabilities. This similarity can be made more precise as follows. First, if an abstract scheduler τ assigns a zero probability to a trace t, then this means that t cannot happen (e.g., the system stops due to overheating). Thus, even if t is a trace that P can exhibit, we would like σ to assign it a zero probability. Second, assume we have a trace t that can be extended with actions a, b, or c, and an abstract scheduler τ that assigns a non-zero probability to all traces $t; X$, with $X \in \{a, b, c\}$ and $\tau(t)(X) = 1$, i.e., τ does not allow for the system to stop after t. If $t; a$ is a trace that P can exhibit, we would like σ to assign it the same probability as τ. However, if P cannot exhibit that trace, σ should assign it a zero probability. But then σ would be a sub-probability measure, i.e., it would allow for P to halt, whereas τ does not. To solve this problem, we proportionally re-distribute the probabilities that τ assigns to the traces that P can exhibit. These two cases are formalized as follows.

Given an *abstract scheduler* τ over a signature S, and a PIOA P with $sig(P) = S$, we define the *refinement* function $\mathsf{refn}(\tau, P) = \tau'$, where $\tau' : (extern(S))^* \rightarrow \mathsf{SubDisc}(extern(S))$, i.e., a function that maps an abstract scheduler and a PIOA to another abstract scheduler, as follows:
Let $t = t'; a \in (extern(S))^*$ in

- if $t \notin traces(P)$, then $\tau'(t')(\{a\}) = 0$;
- if $\tau(t')(\{a\}) = 0$, then $\tau'(t')(\{a\}) = 0$;
- otherwise, $\tau'(t')(\{a\}) = \frac{\tau(t')(\{a\})}{\big(\tau(t')(A)\big)+\big(1-\tau(t')(extern(S))\big)}$,
  where $A = \{x \in extern(S) \mid t'; x \in traces(P)\}$.

Given an abstract scheduler τ and a PIOA P, standard measure theoretic arguments ensure that if τ together with a finite trace t generate a probability measure $\zeta_{\tau,t}$ on the $\sigma-$field $\mathcal{F}_{P_T}$ generated by cones of traces, so does the abstract scheduler $\mathsf{refn}(\tau, P)$, i.e., it generates a probability measure $\zeta'_{\mathsf{refn}(\tau,P),t}$ on the $\sigma-$field $\mathcal{F}_{P_T}$.

We now formalize the relationship between schedulers and abstract schedulers. Given an abstract scheduler τ over a signature S, and a PIOA P with $sig(P) = S$, a scheduler σ is *derivable* from τ iff σ is a scheduler for P such that for all executions $e \in execs(P)$ the trace distributions of $\epsilon_{\sigma,e}$ are equal to the probability measures of $\mathsf{trace}(e)$ assigned by the refinement of τ on P, i.e., for all executions $e, e'' \in execs(P)$, $\mathsf{tdist}(\epsilon_{\sigma,e})(C_{e''}) = \zeta'_{\mathsf{refn}(\tau,P),\mathsf{trace}(e)}(C_{\mathsf{trace}(e'')})$.

3.4. Running Example Modeled Using PIOA

To illustrate how our framework can be used to model enforcement scenarios we will show how to model the running example introduced in §1 (illustrated in Fig. 2b), using PIOA.

Each client C_i requests to open a file x through an $open_i(x)$ output action. Once he receives a file descriptor through an $fd_i(x)$ input action, he requests to close the file through an $close_i(x)$ action. When he receives an acknowledgment that the file was closed, he enters a state from which he stops requesting access to the file. If, however, he is denied access to the file, he decides probabilistically to either enter a state from which it will request to open the file again, or stop requesting.

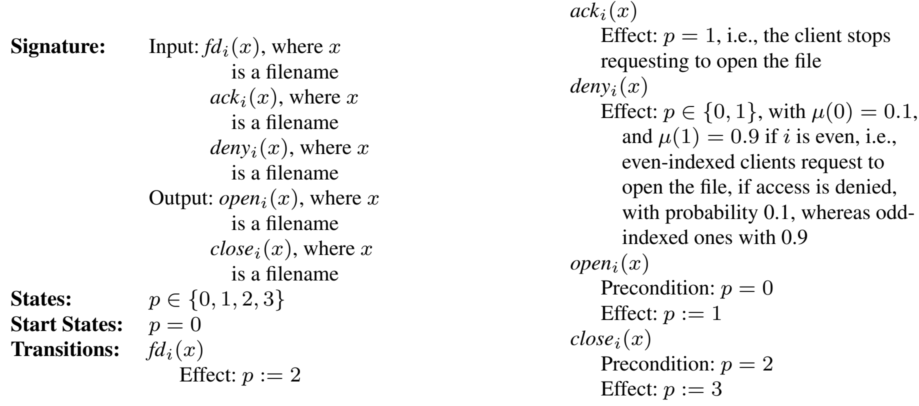


Fig. 3. Client$_i$ PIOA definition

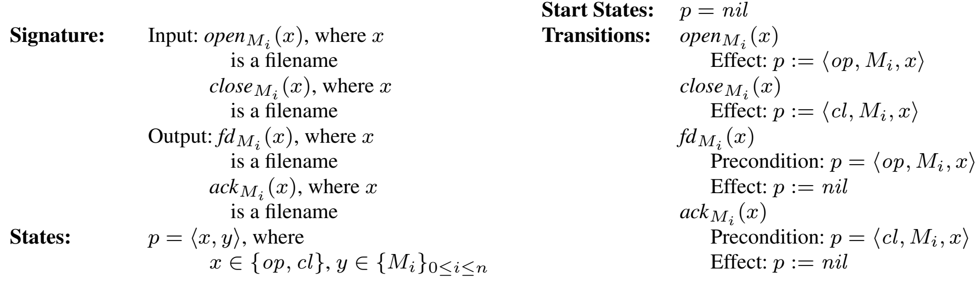


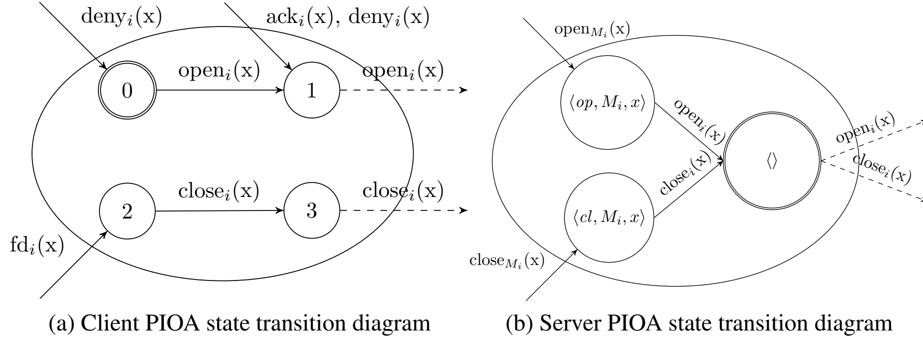
Fig. 4. Server PIOA definition

The pseudocode[3] for C_i is depicted in Fig. 3 and a state diagram in Fig. 5a. The ellipse represents the communication interface of the automaton and the circles the automaton's states. Inputs are depicted as arrows entering the automaton, and we only show the effect of the action, i.e., the automaton's end state. Each output action is depicted with two arrows: (1) a straight arrows between states, to depict the precondition and effect on states; and (2) a dashed arrow to show that action becomes visible outside the automaton. The server S implements a stack of size one: it replies with a file descriptor or an acknowledgment of closing a file for the latest request. This means that if a scheduler allows two requests to arrive before the server is given a chance to reply, then the first request is ignored and the last request is served. The pseudocode for S is depicted in Fig. 4 and a state diagram in Fig. 5b.

To further illustrate some of the capabilities of our framework we introduce two example types of monitor:

- M_{DENY} always denies access to a file that is already open;
- M_{PROB} uses probabilistic information about future requests to make decisions. More precisely, a client i is always granted a request to open a file that is available. Otherwise, if the file is unavailable, i.e., a client j has already opened it, the monitor checks whether (1) after force-closing the file for j, j will ask to re-open the file with probability less than 0.5; and (2) after denying access to i, i

[3]We use the precondition pseudocode style that is typical in I/O automata papers (e.g., [9]).



(a) Client PIOA state transition diagram (b) Server PIOA state transition diagram

Fig. 5. PIOA state transition diagrams

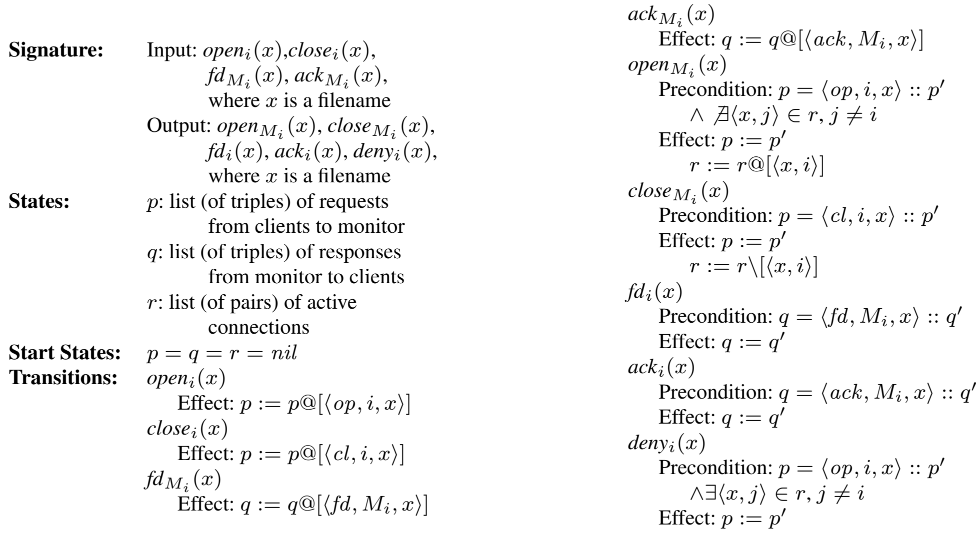


Fig. 6. M_{DENY} PIOA definition

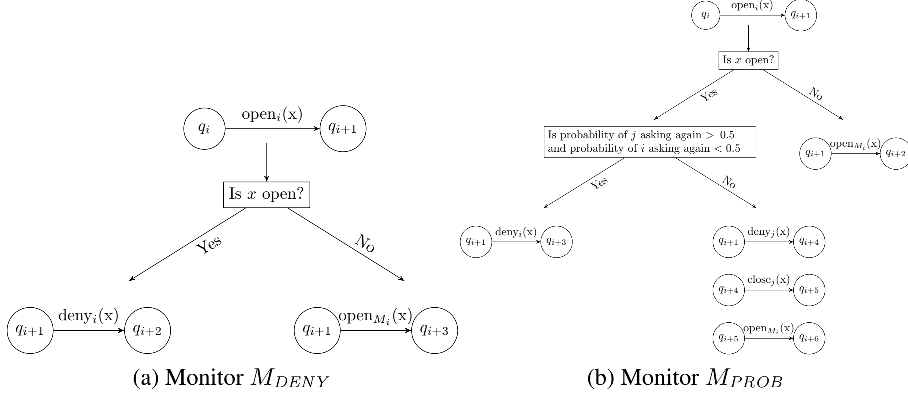
will re-ask with probability greater than 0.5. If both hold, the monitor gives access to i; otherwise it denies access.

The pseudocode for M_{DENY} is depicted in Fig. 6. The pseudocode for M_{PROB} is depicted in Fig. 8. To point out the high level differences in the two monitors with respect to deciding when to accept or deny a request we provide high-level decision diagrams in Fig. 7a and Fig.7b.

Let us now consider the composed system $\Pi = C_1 \times \ldots \times C_n \times M \times S$. The states of the composed system will be $(n+2)-$tuples of the form $q_\Pi = \langle q_{C_1}, \ldots, q_{C_n}, q_M, q_S\rangle$. Two example executions for M_{DENY} are:

- $e_{M_{DENY}} = q_{\Pi_0}\ open_1(x)\ q_{\Pi_1}\ open_{M_1}(x)\ q_{\Pi_2}\ fd_{M_1}(x)\ q_{\Pi_3}\ fd_1(x)\ q_{\Pi_4}\ open_2(x)\ q_{\Pi_5}\ deny_2(x)\ q_{\Pi_6}\ open_2(x)\ q_{\Pi_7}\ deny_2(x)\ q_{\Pi_8}$.
- $e'_{M_{DENY}} = q_{\Pi_0}\ open_1(x)\ q_{\Pi_1}\ open_{M_1}(x)\ q_{\Pi_2}\ fd_{M_1}(x)\ q_{\Pi_3}\ fd_1(x)\ q_{\Pi_4}\ open_3(x)\ q_{\Pi_5}\ deny_3(x)\ q_{\Pi_6}\ open_3(x)\ q_{\Pi_7}\ deny_3(x)\ q_{\Pi_8}$

The corresponding traces are:



(a) Monitor M_{DENY} (b) Monitor M_{PROB}

Fig. 7. Decision Diagrams

Signature: Same as M_{DENY}

States: Same as M_{DENY} and
force $= \langle x, y, z \rangle$, where
$x \in \{0, 1, 2, 3\}, 0 \leq y, z \leq n$

Start States: Same as M_{DENY} and
force $= \langle 0, _, _ \rangle$

Transitions: Same as M_{DENY} with the following changes:

open$_i(x)$
  Effect: if $\exists \langle x, j \rangle \in r, j \neq i$
    $\wedge$ probability of j asking again < 0.5
    $\wedge$ probability of i asking again > 0.5
  then *force* $:= \langle 1, i, j \rangle$
  else $p := p@[\langle op, i, x \rangle]$

open$_{M_i}(x)$
  Precondition: $\big(p = \langle op, i, x \rangle :: p'$
    $\wedge (\nexists \langle x, _ \rangle \in r$
    $\vee force = \langle 3, i, x \rangle\big)$
  Effect: $p := p'$
    $r := r@[\langle x, i \rangle]$
    $force := \langle 0, _, _ \rangle$

close$_{M_i}(x)$
  Precondition: $q = \langle ack, M_i, x \rangle :: q'$
    $\vee force = \langle 2, x, i \rangle$
  Effect: $q := q'$
    $force := \langle 3, x, i \rangle$

ack$_i(x)$
  Precondition: $q = \langle ack, M_i, x \rangle :: q'$
  Effect: $q := q'$

deny$_i(x)$
  Precondition: $(p = \langle op, i, x \rangle :: p'$
    $\wedge \exists \langle x, j \rangle \in r, j \neq i$
    $\vee force = \langle 1, x, i \rangle$
  Effect: $p := p'$
    $force := \langle 2, x, i \rangle$

Fig. 8. M_{PROB} PIOA definition

- $t_{M_{DENY}} = trace(e_{M_{DENY}}) = open_1(x)\ open_{M_1}(x)\ fd_{M_1}(x)\ fd_1(x)\ open_2(x)\ deny_2(x)\ open_2(x)\ deny_2(x)$
- $t'_{M_{DENY}} = trace(e'_{M_{DENY}}) = open_1(x)\ open_{M_1}(x)\ fd_{M_1}(x)\ fd_1(x)\ open_3(x)\ deny_3(x)\ open_3(x)\ deny_3(x)$

In $t_{M_{DENY}}$ client C_1 asks to open file x and he is assigned the file, and after that client C_2 asks to open the same file and is denied access by the monitor. $t'_{M_{DENY}}$ is a similar scenario, with the difference that the client that asks access to x the second time is C_3, i.e., an odd-indexed client.

Let us consider the scheduler σ that schedules transitions based on the following high-level pattern: $\Big([C_1, \ldots, C_n];\ M^*;\ S;\ M^*;\Big)^\omega$. This pattern says that σ chooses probabilistically one of the clients to execute some transition, and then, deterministically, the monitor gets a chance to execute as many actions

as it needs, then the server responds with one transition, and finally the monitor gets again the chance to do as much work as it needs. This pattern repeats finitely, or infinitely, many times.

Let us assume that σ chooses each client to take a turn with probability $P(C_i) = \frac{1}{n}$. Then the probability of $e_{M_{DENY}}$ is given by the measure $\epsilon_{\sigma,\bar{q}}$ on the cone of executions that have $e_{M_{DENY}}$ as prefix, i.e., $\epsilon_{\sigma,\bar{q}}(C_{e_{M_{DENY}}})$. It is easy to calculate that $\epsilon_{\sigma,\bar{q}}(C_{e_{M_{DENY}}}) = \frac{0.1}{n^2}$. We calculate the probabilities of $t_{M_{DENY}}$ and $t'_{M_{DENY}}$ as follows. We know that $\mathsf{tdist}(\epsilon_{\sigma,\bar{q}})(C_{t_{M_{DENY}}}) = \mathsf{trace}(\epsilon_{\sigma,\bar{q}})(C_{t_{M_{DENY}}}) = \epsilon_{\sigma,\bar{q}}(\mathsf{trace}^{-1}(C_{t_{M_{DENY}}}))$. Note that, from the PIOA definitions of (the states of) the components of Π, there is a bijective mapping between the execution $e_{M_{DENY}}$ and the trace $t_{M_{DENY}}$, i.e., the set of executions that trace^{-1} maps $C_{t_{M_{DENY}}}$ to, is the cone $C_{e_{M_{DENY}}}$. Thus, the probability of $C_{t_{M_{DENY}}} = \frac{0.1}{n^2}$. The same holds for calculating the probability of $t'_{M_{DENY}}$.

If we were to consider the execution $e''_{M_{DENY}} = e_{M_{DENY}}\ open_2(x)\ q_{\Pi_5}\ open_{M_2}(x)\ q_{\Pi_6}$, i.e., C_2 tries to access the file again and the monitor gives access to the file, then $\epsilon_{\sigma,\bar{q}}(C_{e''_{M_{DENY}}}) = 0$ (similarly for $t''_{M_{DENY}} = t_{M_{DENY}}; open_2(x)$).

4. Probabilistic Cost of Automata

In this section we develop the framework to reason about the cost of an automaton P.

A cost function assigns a real number to every finite and infinite trace over a signature S, i.e., every possible sequence of external actions of S. More formally:

Definition 1 *A* cost function cost *over a signature S is a signed measure on the σ–field $\mathcal{F}_{\tilde{P}_T}$ generated by cones of traces of an automaton $\tilde{P}$ with $sig(\tilde{P}) = S$ that generates all possible traces over its signature*[4]*, i.e.,* $\mathsf{cost} : \mathcal{F}_{\tilde{P}_T} \rightarrow [-\infty, \infty]$.

Remember that a cone C_t of a finite trace t is the set of traces that have t as prefix. Thus, there is a one-to-one correspondence between traces and the cones (of traces) they infer. Although traces are the subject of our analysis, cones are their (sound) mathematical representation.

Note that in this paper we use the word cost to refer to both expenses and profits. For instance, one can use positive values of costs to possibly represent something you pay (i.e., an expense) and negative values of costs to represent something you gain (i.e., profit). The use of the word when describing practical examples will be clear from the context.

We calculate the expected cost of a trace, called *probabilistic cost*, by multiplying the probability of the trace with its cost. More formally:

Definition 2 *Given a scheduler σ and a cost function* cost*, the* probabilistic cost of a cone of a trace C_t *is defined as* $\mathsf{pcost}_\sigma(C_t) = \big(\epsilon_{\sigma,\bar{q}}(\mathsf{trace}^{-1})(C_t)\big)\mathsf{cost}(C_t)$.

Probabilistic costs of traces can be used to assign expected costs to automata: the probabilistic (i.e., expected) cost of an automaton is the set of probabilistic costs of its traces. However, it is often useful for the cost to be a single number, rather than a set. For example, we might want to build a monitor that does not allow a system to overheat, i.e., it never goes above a threshold temperature. In this case the cost of an automaton (e.g., the composition of the monitor automaton with the system automaton) could be

[4] Given a signature S one can construct such an automaton $\tilde{P}$ by using a single state and a self-looped labeled transition for each action in the signature.

the maximal cost of all traces. Similarly, we might want to build a monitor that "cools down" a system, i.e., lowers a system's temperature below a threshold, infinitely often. Here we could assign the cost of an automaton to be the minimal cost that appears infinitely often in its (infinite) set of traces, and check whether that cost is smaller than the threshold. It is clear that it can be beneficial to abstract the function that maps sets of probabilistic costs of traces to single numbers. We formalize this as follows.

Definition 3 *Given a scheduler σ and a cost function* cost*, the* probabilistic cost of a PIOA P *is defined as* $\mathsf{pcost}^{\mathbb{F}}_{\sigma}(P) = \mathbb{F}_{t \in traces(P)}(\mathsf{pcost}_{\sigma}(C_t))$.

Note that the definition is parametric in the function $\mathbb{F}$.

As an example, consider the infinite set $v = \{v_0, v_1, \ldots\}$, where each v_i is the probabilistic cost of some trace of P (ranging over a finite set of possible costs); then, $\mathbb{F}$ could be (following definitions of Chatterjee et al. [12]):

- $\mathsf{Sup}(v) = sup\{v_n \mid n \geq 0\}$, or
- $\mathsf{LimInf}(v) = liminf_{n\to\infty} v_n = lim_{n\to\infty} inf\{v_i \mid i \geq n\}$.

Sup chooses the maximal number that appears in v (e.g., the maximal temperature that a system can reach). LimInf function chooses the minimal number that appears infinitely often in v (e.g., the temperature that the system goes down to infinitely often).

For the rest of the paper we will assume that $\mathbb{F}$'s arguments are sets of pairs $\langle t, c\rangle$, where the first component t is a trace and the second component c is (typically) the expected cost of t. This assumption will simplify the presentation and analysis of our results.

Note that if we were to assign costs to actions r_1 and r_2, e.g., 2 and 5 respectively, then cost can assign different numbers to their interleavings that might not clearly relate to the costs of the actions, e.g., $\mathsf{cost}(r_1; r_2) = 0$ and $\mathsf{cost}(r_2; r_1) = 20$.

Next, we show how one can define the cost of a system given cost functions for its components.

Assume that we have cost functions for the clients, the monitor, and the server (resp. cost_{C_i}, cost_M, and cost_S). The *cost of a trace t of the system* Π (i.e., the composition of the clients, monitor, and server) is the sum of the probabilistic costs of each component, whenever that component is allowed to take a step in the trace.

More formally[5]:

- $\mathsf{Cost}(C_\lambda)^6 = \mathsf{cost}_{C_i}(C_\lambda) + \mathsf{cost}_M(C_\lambda) + \mathsf{cost}_S(C_\lambda)$.
- $\mathsf{Cost}(C_t) = \mathsf{Cost}(C_{t'}) + \mathit{diff}_X(C_{t';a})$, where $X \in \{C_i, M, S\}$, $t = t'; a$, $a \in acts(X)$, and $\mathit{diff}_X(t'; a) = \mathsf{cost}_X(C_{t';a}) - \mathsf{cost}_X(C_{t'})$.

The probabilistic cost of a trace and the probabilistic cost of an automaton are now defined, as above, using the cost function Cost.

One can alternately define the cost of a system based on costs assigned to smaller components. For example, we can define cost functions over individual actions (or transitions) and use them to define the cost of a trace, and the cost of an automaton. In this case, the cost of traces would be independent of the interleaving of transitions (unless we define costs over transitions and use some coding trick, e.g., allow states to encode the trace history). This approach is similar to the definition of Cost so we do not pursue it here. Note that such an approach can be used to embed the framework of Drabik et.al. in ours [16].

[5] We use addition as the function between costs of actions in the trace. One can define appropriate cost functions by using other functions $\mathbb{G}$ from costs to costs, but we do not pursue it in this paper.

[6] Remember, λ denotes the empty trace.

5. Cost Security Policy Enforcement

In this section we define security policies and what it means for a monitor to enforce a security policy on a system.

Cost security policies. A monitor M is a PIOA. A monitor mediates the communication between system components S_i which are also PIOA. Thus, the output actions of each S_i are inputs to the monitor, and the monitor has corresponding outputs that it forwards to the other components. More formally, given an index set I and a set of components $\{S_i\}$, $i \in I$, we assume that $acts(S_i) \cap acts(S_j) = \emptyset$, for all $i, j \in I$, $i \neq j$. Our goal is to model and reason about the external behavior of the monitored system. Thus, we also assume that $internal(S_i) = \emptyset$, for all $i \in I$. Since the system components S_i are compatible, we will refer to their composition $\Pi_{i \in I} S_i$ as system S. A monitored system is the PIOA that results from composing M with S.[7]

The cost function defined in §4 describes the impact of a monitor on a system. A cost function is not necessarily bound to a specific security policy, which allows for the analysis of the same monitor against different policies. In practice, a monitor's purpose is to ensure that some policy is respected by the monitored system. In the running example, the monitor's role is to ensure that a file is not simultaneously opened by two clients. Furthermore, since each $deny$ action comes with a cost, it is desirable for the cost of monitoring to be limited. This motivates the need to define a cost security policy.

Definition 4 *Given a (monitored) system* $(M \times S)$*, a* cost security policy Pol *over* $sig(M \times S)$ *is a cost function over* $sig(M \times S)$*, i.e., a signed measure* Pol *on the* σ*−field* $\mathcal{F}_{\tilde{P}_T}$[8] *generated by cones of traces of the system, i.e.,* $\mathsf{Pol} : \mathcal{F}_{\tilde{P}_T} \rightarrow [-\infty, \infty]$.

When we talk about the signature, actions, etc. of Pol, we refer to the signature, actions, etc of P. Cost security policies associate a cost with each trace. For instance, if a trace t corresponds to a particular enforcement interaction between a monitor and a client, then $\mathsf{Pol}(C_t) = 10$ could describe that such enforcement (i.e., t) is allowed only if its cost is less than 10. Our definition of policies extends that of security properties [36]: security properties are predicates, i.e., binary functions, on sets of traces, whereas we focus on policies that are functions whose range is the real numbers (as opposed to $\{0, 1\}$). We leave the investigation of enforcement for securities policies defined as sets of sets of traces (e.g., [36, 14,29]) for future work.

Definition 5 *Given a cost security policy* Pol *and a scheduler* σ *the* probabilistic cost security policy pPol_σ *under* σ *is defined as* $\mathsf{pPol}_\sigma(C_t) = \big(\epsilon_{\sigma,\bar{q}}(\mathsf{trace}^{-1})(C_t)\big)\mathsf{Pol}(C_t)$.

Cost security policy enforcement. In §4 we showed how to calculate the expected cost of a trace of an automaton, and the expected cost of an automaton. In the previous paragraph we defined the notion of a (probabilistic) cost security policy. We will now define what does it mean for a monitor to enforce a cost security policy on a system.

[7]By assumption, M and S are compatible. In scenarios where this is not the case, one can use renaming to make the automata compatible [29,9,10].

[8]Remember $\tilde{P}$ is the automaton that has the same signature as $(M \times S)$, in this case, and produces all possible traces over its signature.

Definition 6 *Given a scheduler σ, a cost function* cost*, a policy* Pol*, a function $\mathbb{F}$, a monitor M, and a system S (compatible with M), we say that M $n-$enforces$_{\leq}$ (resp., $n-$enforces$_{\geq}$)* Pol *on S under σ, $\mathbb{F}$, and* cost *if and only if the probabilistic cost of the monitored system differs by at most n from the probabilistic cost that the policy assigns to the traces of the monitored system, i.e.,:*

$\left(\mathsf{pcost}^{\mathbb{F}}_{\sigma}(M \times S)\right) - \left(\mathbb{F}_{t \in traces(M \times S)} \mathsf{pPol}_{\sigma}(C_t)\right) \leq n$ *(resp., $\geq n$), i.e.,*

$\left(\mathbb{F}_{t \in traces(M \times S)} \mathsf{pcost}_{\sigma}(C_t)\right) - \left(\mathbb{F}_{t \in traces(M \times S)} \mathsf{pPol}_{\sigma}(C_t)\right) \leq n$ *(resp., $\geq n$).*

We say that a monitor M *enforces*$_{\leq}$ (resp., *enforces*$_{\geq}$) a security policy P on a system S under a function $\mathbb{F}$, a scheduler σ, and a cost function cost if and only if M $0-$enforces$_{\leq}$ (resp., $0-$enforces$_{\geq}$) P on S under $\mathbb{F}$, σ, and cost.

The definition of enforcement says that a monitor M enforces a policy Pol on a system S if the probabilistic cost of the monitored system under some scheduler σ and cost function cost is less or equal (resp. greater or equal) than the cost that the policy assigns to the behaviors that the monitored system can exhibit. We define enforcement using two comparison operators because different scenarios might assign different semantics to the meaning of enforcement: One might use a monitor to maximize the value of a monitored system with respect to some base value, e.g., in our running example, we may want to give access to as many unique clients as possible since the server is making extra money by delivering advertisements to them; thus, the monitor has motive to give priority to every new request for accessing a file. In other cases, one might use a monitor to minimize the cost of the monitored system with respect to some allowed cost, e.g., we might want to minimize the state that the monitor and the server keep to provide access to files, in which case caching might be cost-prohibitive. Without loss of generality in this paper we focus on $\leq$; similar results hold for $\geq$.

Enforcement is defined with respect to a global function $\mathbb{F}$. $\mathbb{F}$ transforms the costs of all traces of a monitored system to a single value. As described in §4, this value could represent the minimum cost of all traces, their average, sum, etc. Thus, $\mathbb{F}$ can model situations where an individual trace might have cost that is cost-prohibited by the policy (e.g., overheating temporarily), but the monitored system as a whole is still within the acceptable range (i.e., before and after the overheating the system cools down enough).

In the previous instantiation of our running example, there might exist some trace t where $\mathsf{cost}(t) > \mathsf{Pol}(t) > -\infty$, typically when a client keeps asking for a file that is denied. Although this would intuitively mean that the cost security policy is not respected for that particular trace, it might be the case that M enforces Pol, as long as Pol is globally respected, which could happen, e.g., if the probability of t is small enough. This illustrates a strength of our framework: we can allow for some local deviations, as long as they do not impact the global properties, i.e., overall expected behavior, of the system. If we wish to constrain each trace, we can define *local enforcement*, which requires that the cost of *each trace* of the monitored system is below (or above) a certain threshold, as opposed to enforcement which requires that the value of some function computed over *all traces* of the monitored system is below (or above) a certain threshold. Note that local enforcement can be expressed through a function $\mathbb{F}$ that universally quantifies the cost difference from the threshold over all traces of the monitored system. Local enforcement could be useful, for example, to ensure that a system *never* overheats even momentarily, whereas enforcement would be useful if we want to have probabilistic guarantees of the system; e.g., we accept a 0.001% probability that the system will become unavailable due to overheating.

A question a security designer might have to face is whether it is possible, given a boolean security policy that describes what should not happen and a cost policy that describes the maximal/minimal allowed cost, to build a monitor that satisfies both. This problem can help illuminate a common cost/security tradeoff: the more secure a mechanism is, the more costly it usually is.

There is a close relationship between boolean security policies (e.g., [36]) and cost security policies: given a boolean security policy there exists a cost security policy such that if the cost security policy is $n-$enforceable then the boolean security policy is enforceable as well (and vice versa). Specifically, given a boolean security policy P, we write Pol_P for the function such that $\mathsf{pPol}_P(C_t) = 0$ if $P(t)$ holds, and $-\infty$ otherwise. Given a predicate P, if we instantiate function $\mathbb{F}$ with the function that returns the least element of a set and function cost with the function that maps every (trace) cone to 0, and if M $0-$enforces$_\leq$ Pol_P, then any trace belongs to P. In other words, our framework is a generalization of the traditional enforcement model.

In the other direction, since cost security policies are more expressive than boolean security policies, we need to pick a bound that will serve as a threshold to classify traces as acceptable or not. Given a probabilistic cost security policy pPol, a cost function cost, a scheduler σ and a bound $n \in \mathbb{R}$, we say that a trace t *satisfies* $\mathsf{Pol}_{\mathsf{cost},n,\sigma}$, and write $\mathsf{Pol}_{\mathsf{cost},n,\sigma}(t)$ if and only if $\mathsf{pPol}(C_t) \geq \mathsf{pcost}_\sigma(C_t) - n$.

Expressing cost security policies as boolean security policies allows one to embed in our framework a notion of sound enforcement [25]: a monitor is a sound enforcer for a system S and security policy P if the behavior of the monitored system obeys P. As described above, one encodes P in our framework as Pol_P, which returns $-\infty$ if a trace violates P and 0 otherwise. Sound enforcement can be expressed as $0-$enforcement$_\leq$ using a global function $\mathbb{F}_P$ that assigns $-\infty$ to the cost of the automaton composition that represents the monitored system if some trace has cost $-\infty$, and 0 otherwise. Specifically, if a monitor soundly enforces P on a system, all its traces will belong to P and Pol_P will map them all to 0, which when applied to $\mathbb{F}_P$, will result in a global cost of 0. If the monitor is not sound, then the global cost will be $-\infty$. Thus, a monitor soundly enforces a boolean security policy P if and only if the monitor $0-$enforces$_\leq$ the cost security policy Pol_P under $\mathbb{F}_P$ and $\mathsf{cost}(_) = 0$.

Transparent cost enforcement. Assume that we are given a scheduler σ, a cost function cost, a function $\mathbb{F}$, a policy Pol, and a system S and we want to $n-$enforce$_\leq$ Pol on S under σ, $\mathbb{F}$, and cost. As we explained in the introduction, different monitors may be able to achieve this goal. One such choice might be a monitor that acts as a "sink": it never forwards messages between the system's components. Thus,interaction-intensive systems, i.e., systems that rely heavily on interaction to achieve their goals, will only exhibit a minimal behavior. As such, the cost of such a monitored system will be (close to) zero (assuming that components limit their non-interactive actions–cf. *quiescent forgiveness* in [29]). However, such monitors are unlikely to be useful in practice.

Previous work on run-time enforcement has identified (similar) situations where trivial monitors enforce (boolean) policies by consuming all inputs and denying all actions the target system wants to execute [18,24,22]. *Transparency* is one notion that aids in ruling out such uninteresting cases: if the target system wants to perform an action that obeys the policy, then the monitor must allow it. Most definitions of transparency that have been introduced so far are within frameworks where policies reason only about the target's behavior [36,25]: a policy is a predicate over traces of the target (i.e., a subset of the traces that the target might exhibit) and not over traces that the monitored target can exhibit through the interaction of the monitor with the target.

In this paper we take a (more general) view, that has been recently introduced, which allows policies to describe how monitors are allowed to react to target's requests [26,29] (in addition to considering policies which reason about cost). Thus, enforcement is now implicit in the definition of a policy (i.e., in the traces that the policy allows). This means that we can define transparency as a specific type of interaction between the target and the monitor.

Since our definition of policies is more expressive than previous ones with respect to the interaction between the target and the monitor, to talk about transparent enforcement we first need to encode previous

definitions of policies (i.e., sets of traces by a target) in our framework [28]. The main idea is as follows: given a policy that describes which target's traces are allowed, we build a policy (over the monitored target) in which every valid trace of the target is "forwarded" by the monitor to the environment (and vice versa). The way that the "forwarding" can be achieved by the monitor depends on the notion of fairness that we assume in the model: if the monitor is not allowed to finish forwarding the valid trace of the target then it will not achieve transparent enforcement, even though the same monitor under different circumstances would be transparent. Thus, we need to choose a notion of transparency depending on whether we assume fairness in our model or not [28]. In the framework of this paper (weak) fairness can be encoded by schedulers that do not assign zero probabilities to steps of an automaton from a given state (if such steps are defined in the transition relation of the automaton). Once we have encoded (boolean) policies of targets and a notion of transparent enforcement as a (boolean) policy over monitored targets, we can use the translation described previously to derive a cost policy and talk about cost transparent enforcement. This process has some technical nuances but it is straightforward to implement and we will not pursue it more in this paper.

6. Cost Comparison

Given a system S, a function $\mathbb{F}$, a scheduler σ and a monitor M, $\mathsf{pcost}^{\mathbb{F}}_{\sigma}(M)$ and $\mathsf{pcost}^{\mathbb{F}}_{\sigma}(M \times S)$ are values in $[-\infty, \infty]$, and as such provide a way to compare monitors.

To meaningfully compare monitors, we need to fix the variables on which the cost of a monitor depends, i.e., functions $\mathbb{F}$ and cost, and the scheduler σ. Difficulties arise when trying to fix a scheduler for two different monitors (and thus monitored systems), even if they are defined over the same signature. States of the monitors, and thus their executions, will be syntactically different and we cannot directly define a single scheduler for both. Moreover, since schedulers assign probabilities to specific PIOA and their transitions, one scheduler cannot be defined for two different monitors.

To overcome this difficulty we rely on the abstract schedulers introduced in §3.3. Namely, to compare two monitored systems we use a single abstract scheduler which we then refine into schedulers for each monitored system.[9]

Abstract schedulers allow us to "fairly" compare two monitors, but additional constraints are needed to eliminate impractical corner cases. To this end we introduce *fair abstract schedulers*.

Definition 7 *An abstract scheduler τ over the signature of a class of monitored targets $\mathcal{M} \times \mathcal{S}$ is* fair *(w.r.t. comparing monitors) if and only if (1) if there are infinitely many traces t of a monitored target with an extension t' such that $t' = t; a$ where $a \in \mathit{output}(M)$, then infinitely many of these extensions t' are assigned a non-zero probability by τ, and (2) for every trace t of a monitored target, every extension t' of t by a monitor's actions, i.e., $t' = t; a$ with $a \in \mathit{extern}(M)$, is assigned the same probability by τ.*

Constraint (1) ensures that a fair abstract scheduler will not starve the monitor, i.e., the monitor will always eventually be given a chance to enforce the policy. We do not impose the stronger requirement that the monitor intervenes at each step that the target takes. This is because, in the context of comparing two monitoring strategies that enforce a security policy over a given target, the requirement for the monitor

[9] An abstract scheduler τ also provides a meaningful way to compare monitors with different signatures: calculate the union S of the signatures of the two monitors and (1) use a τ with signature S, and (2) extend each monitor's signature to S. This is useful when comparing monitors of different capabilities, e.g., a truncation and an insertion monitor [24], where the insertion monitor might exhibit additional actions, e.g., logging.

to intervene at each step would exclude some potentially cost-efficient, but still sound, strategies. For example, assume that checking a target's security relevant action incurs a fixed cost. If a given target can misbehave only on every other security relevant action, then a monitor that checks every target's action is going to incur a higher cost than a monitor that only checks every other action. For that particular target, the second monitor can still enforce the security policy.

Constraint (2) ensures that the abstract scheduler is not biased towards a specific monitoring strategy. For example, an unfair scheduler could assign zero probability to arbitrary monitoring actions (e.g., the scheduler "stops" insertion monitors [24]) and non-zero probability to monitors that output "valid" target actions verbatim (i.e., the scheduler allows suppression monitors [24]). Such a scheduler would be unlikely to be helpful in performing a realistic comparison of the costs of enforcement of an insertion and a suppression monitor. Note, that an abstract scheduler could still assign zero probabilities to all possible monitors. This allows us to model highly adversarial scenarios, such as the case where the attacker has control over the operating system and can stop the execution of a monitor at will.

There might be scenarios where such schedulers are appropriate[10], but in this paper we pursue only the equiprobable scenario.

Note that, in our model, there are two ways for adversarial behavior to appear. First, the attacker is one of the clients, i.e., the attacker provides inputs to the monitor. In principle, the attacker could attempt to drive the higher-cost behavior of a program through carefully crafted inputs. But, in this paper we do not consider an explicit, active attacker. Instead, we consider the cost of a monitor that interacts with many clients, each of which has the potential to perform some actions that would violate the security policy if there was no monitor to prevent that. By modeling the clients (i.e., the environment of the monitor) probabilistically, we represent a mixed population of clients, consisting of: honest clients that never try to perform any non-secure action; honest clients which might attempt to perform a non-secure action by mistake; active attackers, which might try to perform only non secure actions; and clients which might act differently in different contexts. Of course, an active attacker might cause a huge cost on its own, forcing the monitor to keep performing costly security actions, but, in practice, such a client is balanced by a secure client, whose actions have no cost for the monitor. Hence, in this paper, we present a framework that allows to model the overall expected cost of enforcement, rather than on the worst case scenario, which would assume that each client is an active attacker. Note that modeling the worst case is still expressible in our framework (by modeling each client as an attacker).

Second, the attacker could affect the scheduler (or equivalently, the scheduler could be adversarial). In this case we have made two assumptions. First, the monitor designer "knows" the attacker's strategy. This is equivalent to stating that the monitor designer designs the monitor for defending against specific attacks. This is typical in the context of security, where a defense is proven secure against a specific attacker or threat model. Second, we assume that the attacker cannot arbitrarily stop the monitor from intervening (Def. 7). If that was the case, then the monitor would be unable to enforce a policy, even though in principle it could.

Definition 8 *Given a system S, a function $\mathbb{F}$, a cost function* cost, *two monitors M_1 and M_2 with $sig(M_1) = sig(M_2)$, an abstract scheduler τ over $sig(M_1 \times S)$, two schedulers σ_1 (for $M_1 \times S$) and σ_2 (for $M_2 \times S$) derivable from τ, and a well-order $\trianglelefteq$ we say that M_2 is* less costly *than a monitor M_1 and write $M_2 \trianglelefteq M_1$, if and only if* $\mathsf{pcost}^{\mathbb{F}}_{\sigma_2}(M_2 \times S) \trianglelefteq \mathsf{pcost}^{\mathbb{F}}_{\sigma_1}(M_1 \times S)$.

[10]This is a similar situation with having various definitions for fairness [23].

Running example. Typically, when a monitor modifies the behavior of the system some cost is incurred (e.g., the usability of the system decreases, computational resources are consumed). For instance, building on the running example (§3.4), one way monitors can modify the behavior of the system is by denying an access to a client.

Let us assume that each deny action incurs a cost of 1. Then we can define a function cost_D that associates with each trace the cost n, where n is the number of denies that appear in the trace. Moreover, let us assume that (1) $\mathbb{F}$ is Sup, and (2) the abstract scheduler τ follows the pattern $\Big([C_1, \ldots, C_n];\ M^*;\ S;\ M^*\Big)^\infty$ as described in §3.4. Assuming we have two clients C_1 and C_2, our monitored system is $\Pi = C_1 \times C_2 \times M \times S$. If M is M_{DENY}, then we refine τ to the scheduler $\sigma_{M_{DENY}}$; dually, the scheduler for M_{PROB} will be $\sigma_{M_{PROB}}$. The probabilistic cost of the monitored system with M_{DENY} is $\mathsf{sup}_{t \in traces(\Pi_{M_{DENY}})}(\mathsf{pcost}_{\sigma_{M_{DENY}}}(C_t))$, and similarly for M_{PROB}.

We observe that if we assume that C_1 and C_2 ask for a file after a denied request with probability p_1 and p_2 respectively, with $p_1 < 0.5$ and $p_2 > 0.5$, then C_1 is less likely to ask again for a file which has been denied. In this case, it is better to deny an access to C_1 rather than to C_2, in order to limit the number of deny actions. Hence, with such a system, we have $M_{PROB} \leq M_{DENY}$.

Also, observe that the last result is sound only under the assumption that schedulers $\sigma_{M_{DENY}}$ and $\sigma_{M_{PROB}}$ are compatible with τ. If that was not the case, then $\sigma_{M_{DENY}}$ could starve C_2 (or $\sigma_{M_{PROB}}$ could starve C_1). This would give M_{DENY} an unfair advantage over M_{PROB}, and we would have as a result that $M_{DENY} \leq M_{PROB}$. Such unfair comparisons are ruled out by requiring schedulers to be compatible.

6.1. Optimality

In this section we discuss the existence of cost optimal monitors and the sufficient conditions for constructing cost-optimal monitors.

Definition 9 *A monitor M is* cost optimal *for a system S and a well order $\trianglelefteq$ if and only if for all monitors M' with $sig(M) = sig(M')$, $M \trianglelefteq M'$.*

The next proposition states that for any system S a cost optimal monitor exists.

Proposition 6.1 *Given a system S there is a cost optimal monitor M (for some well-order $\trianglelefteq$).*

Proofs can be found in App. 1.

Note that although for every system S we can find a cost optimal monitor M, M is cost optimal for a specific well-order $\trianglelefteq$ which might not be the standard well-order $\leq$. For example, if the expected costs of monitored systems take values from the natural numbers then there exists an optimal monitor under $\leq$. On the other hand, if the expected costs of monitored systems range over the integers, or real numbers, then there might not exist a cost optimal monitor under $\leq$ (although a cost optimal monitor might exist for a different well order $\trianglelefteq$[11]).

Proposition 6.1 guarantees that a cost-optimal monitor exists (for given system S, function $\mathbb{F}$, cost function cost, and abstract scheduler τ). However, we might not be able to find or construct such a cost-optimal monitor. One reason for this inability can be, for example, that $\mathbb{F}$ is not be differentiable.

[11]For instance, since there are countably many monitors, one can use a bijection that maps monitors to natural numbers and the standard well-ordering of the natural numbers.

However, for many practical purposes functions $\mathbb{F}$ for which we can find optimal monitors will be used. One such class of functions is *monotone* functions. Monotone functions map arguments to values in such a way that they preserve some order, i.e., if two arguments are ordered under some ordering, then so are the corresponding values when the function is applied on the arguments. For instance, the function that maps traces of a system to CPU cycles or power consumption is monotone: the more time the system runs, i.e., the longer the trace of the system, the more CPU cycles and power are consumed. Another example is the function that maps valid executions of a monitor that mediates communication between a client and a server to profits for the server: the longer a valid execution, the more profit the server will make. This function is relevant in scenarios where some Service Level Agreement exists between a client and a server, and we want to maximize the well-behaved interaction amongst them (cf. transparency–§5). Next we formally define monotone functions $\mathbb{F}$ (remember that we assume that $\mathbb{F}$'s arguments are sets of pairs $\langle t, c\rangle$, where the first component t is a trace and the second component c is (typically) the expected cost of t).

Definition 10 *Given two sets X, Y of pairs of traces and real numbers, i.e., $X, Y \in 2^{\Sigma^\infty \times \mathbb{R}}$, we write $X \sqsubseteq Y$ if and only $\forall \langle t_1, c_1\rangle \in X : \exists \langle t_2, c_2\rangle \in Y : \langle t_1, c_1\rangle \sqsubseteq \langle t_2, c_2\rangle$, where $\langle t_i, c_i\rangle \sqsubseteq \langle t_j, c_j\rangle$ if and only if $t_i \preceq t_j$ and $c_i \trianglelefteq c_j$.*

We say that a function $\mathbb{F} : 2^{\Sigma^\infty \times \mathbb{R}} \to \mathbb{R}$ is monotone *if and only if it is monotone under the ordering $\sqsubseteq$, i.e., if $X \sqsubseteq Y$ then $\mathbb{F}(X) \trianglelefteq \mathbb{F}(Y)$.*

The next theorem formalizes the intuition that when dealing with monotone functions we can exploit knowledge about the scheduler and the cost function to build cost optimal monitors.

Theorem 6.1 *Given:*

1. *a finite-state system S,*
2. *a cost assignment* map *of each action of S to a real value such that for every input action i of S,* $\mathsf{map}(i) = 0$,
3. *a cost function* cost *that is defined recursively based on* map *such that the* cost *of a trace t is the sum of the values of the actions that appear on t,*
4. *a weakly fair*[12] *abstract scheduler τ, and*
5. *a function $\mathbb{F}$ that is monotone and continuous (i.e., it preserves limits),*

we can construct a cost optimal monitor for the standard ordering $\leq$ of real numbers.

Thm. 6.1 provides a generic description of the conditions sufficient for constructing a cost-optimal monitor. In the constructive proof of Thm. 6.1 we try to find a monitor M such that the (finite-state) monitored system $(M \times S)$ contains as many cycles as possible. The cycles together with the constraint that the scheduler is weakly fair guarantee the existence of infinite traces. The monotonicity of $\mathbb{F}$ and the "recursive additive" definition of the cost function cost guarantee that the monitor that contains the most highest-value cycles will be cost optimal. One might argue that Thm. 6.1 contains many (severe) restrictions. However, this is something unavoidable: there is no generic algorithm that will produce a

[12] As described in §5, weakly fair schedulers are the schedulers that do not assign zero probabilities to steps of an automaton from a given state (if such steps are defined in the transition relation of the automaton).

cost-optimal monitor for every (arbitrary) choice of functions and targets[13]. On the other hand, a positive interpretation of Thm. 6.1, based on the previous discussion about monotone functions, is that in (many) practical applications we have to deal with "simple" enough functions, and thus we can construct (verifiable) optimal monitors.

Running example. We extend the running example from the previous subsection and §3.4. We assume that each deny action incurs a cost of 1. We define the function cost_P that associates with each trace the cost $\sum_n \frac{1}{n^2}$, where n is the number of denies that appear in the trace. Also, if a trace gives access to a file twice without closing it first, then cost_P assigns to the trace the value $-\infty$. This means that violating the security policy is always worse than any other decision. Moreover, let us assume that (1) $\mathbb{F}$ is Sup, and (2) that the weakly fair abstract scheduler τ follows the pattern $\Big([C_1, \ldots, C_n];\ M^*;\ S;\ M^*\Big)^\infty$ as described in §3.4. Note that these assumptions satisfy constraints (2), (3), (4), and (5) of Thm. 6.1.

Now assume we have three clients C_1, C_2, and C_3. Our monitored system is $\Pi = C_1 \times C_2 \times C_3 \times M \times S$. If M is M_{DENY}, then we refine τ to the scheduler $\sigma_{M_{DENY}}$; dually, the scheduler for M_{PROB} will be $\sigma_{M_{PROB}}$. The probabilistic cost of the monitored system with M_{DENY} is $\mathsf{sup}_{t \in traces(\Pi_{M_{DENY}})}$ $(\mathsf{pcost}_{\sigma_{M_{DENY}}}(C_t))$, and similarly for M_{PROB}.

Let us assume that C_1, C_2, and C_3 have finite states, and that they ask for a file after a denied request with probability $p_1 = 0.1$ $p_2 = 0.5$, and $p_3 = 0.6$. Note that this assumption satisfy constraint (1) of Thm. 6.1 and we can construct an optimal monitor. The optimal monitor gives priority to C_3 over C_2 and C_1, and also gives priority to C_2 over C_1.

It is worth noting that, with such a system, we still have $M_{PROB} \leq M_{DENY}$ (as in the previous subsection). However, M_{PROB} is not an optimal monitor. Assume C_2 has accessed the file, and now C_3 also requests to access it. Since the probability of C_2 is not less than 0.5, M_{PROB} will not close the file and give access to C_3.

One important point is that cost_P assigns $-\infty$ to traces that violate the policy, whereas the cost function cost_D from the previous section, does not. If we were to use the cost function cost_D when looking for an optimal monitor, then the maximal (i.e., best) reachable cost is 0, meaning that no deny action should be returned. It follows that the cost-optimal monitor never denies any action, and, clearly, this monitor does not generally respect the requirement that at most one client at a time should have access to a particular file. Thus, care must be taken when defining cost functions and reasoning about optimal monitors enforcing a security policy.

$N-$step optimality. In practice, the risk (cost-benefit) analysis is a dynamic process: probabilities for attacks to happen (arrival of inputs) are re-evaluated based on the current state of the system (e.g., sudden publicity, or increase in the value of the company), costs might have changed (e.g., the state of the art cryptographic protocols might require more computational resources), and such changes affect the optimality of an enforcement system. Thm. 6.1 gives us a generic description of what conditions are sufficient for constructing a cost-optimal monitor.

However, the assumption of having schedulers and cost functions that have valid information about the future might be too ambitious. Next, we present a special case of the theorem where we only have

[13] For example, let us assume that targets are infinite-state systems (i.e., typical programs) that output their own descriptions (i.e., source code). Also, assume that monitors receive the descriptions of targets and output the descriptions of other programs (i.e., they transform programs from one form to another). Finally, assume that our cost policy is that the monitor outputs the smallest program with the same behavior as the target. In this case, an algorithm to construct a cost optimal monitor would amount to a"fully optimizing compiler", which is known to be undecidable [1].

information about $n-$steps ahead in the future, where $n \in \mathbb{N}$. Before we state the modified theorem, we need to first adjust our basic definitions so they reason about the partial knowledge that we may have.

Given a scheduler σ intuitively we construct a $n-$step scheduler σ_n by restricting the knowledge that σ may have about the probabilities of actions to happen after an execution that has length at most n. More formally:

Definition 11 *Given a scheduler σ, an $n-$step scheduler σ_n is a function $\sigma_n : execs^*(P) \to \mathsf{SubDisc}(R(P))$ defined as:*

- $\sigma_n(e) = \sigma(e)$, *if* $|trace(e)| \leq n$,
- $\mathsf{supp}(\sigma_n(e)) = \emptyset$, *otherwise.*

Note that a $n-$step scheduler σ_n is still a scheduler but with "less information than the original scheduler σ. Thus, the definition of a measure of cone and trace remain the same as in §3.2.

The definition of an $n-$step abstract scheduler follows that of a $n-$step scheduler:

Definition 12 *Given an abstract scheduler τ over a signature S, an $n-$step abstract scheduler τ_n is a function $\tau_n : (extern(S))^* \to \mathsf{SubDisc}(extern(S))$ defined as:*

- $\tau_n(t) = \tau(t)$, *if* $|t| \leq n$,
- $\mathsf{supp}(\tau_n(t)) = \emptyset$, *otherwise.*

Given a cost function cost, a $n-$step cost function $\mathsf{cost_n}$assigns a real number to every trace over a signature S that has length at most n, and 0 otherwise. More formally:

Definition 13 *Given a cost function* cost *over a signature S, a $n-$step cost function* $\mathsf{cost_n}$ *over S is defined as*

- $\mathsf{cost_n}(t) = \mathsf{cost}(t)$, *if* $|t| \leq n$,
- $\mathsf{cost_n}(t) = 0$, *otherwise.*

We calculate the $n-$step probabilistic cost of a trace as described in §4, i.e., by multiplying the probability of the trace with its cost. Note that when a trace has length larger than n then its $n-$step probabilistic cost will be zero. Although we could have defined the (expected) cost of such a trace to be undefined (e.g., $\perp$) rather than a concrete value, we set up our definitions in such a way that this value does not cause any conflicts when the expected costs of traces with length less than n is also zero. In the alternative case we would have to adjust many definitions and calculations, e.g., those of measures, with significant notational overhead.

Definition 14 *Given a function $\mathbb{F}$, we define the $n-$step restriction of $\mathbb{F}$ to be the restriction of $\mathbb{F}$ to the traces that have length at most n, and write $\mathbb{F}\restriction_n$.*

Definition 15 *Given a system S, a function $\mathbb{F}$, a $n-$step cost function* $\mathsf{cost_n}$*, two monitors M_1 and M_2 with $sig(M_1) = sig(M_2)$, an $n-$step abstract scheduler τ_n over $sig(M_1 \times S)$, two $n-$step schedulers σ_1 (for $M_1 \times S$) and σ_2 (for $M_2 \times S$) derivable from τ_n, and a well-order $\trianglelefteq$ we say that M_2 is $n-$*step less costly *than a monitor M_1 and write $M_2 \trianglelefteq_n M_1$, if and only if* $\mathsf{pcost}_{\sigma_2}^{\mathbb{F}\restriction_n}(M_2 \times S) \trianglelefteq \mathsf{pcost}_{\sigma_1}^{\mathbb{F}\restriction_n}(M_1 \times S)$.

Definition 16 *A monitor M is $n-$*step cost optimal *for a system S and a well order $\trianglelefteq$ if and only if for all monitors M' with $sig(M) = sig(M')$, $M \trianglelefteq_n M'$.*

Now we are ready to state the equivalent of Thm. 6.1 for the case where our information and knowledge is limited for just $n-$steps:

Theorem 6.2 *Given a number $n \in \mathbb{N}$ and:*

1. *a finite-state system S,*
2. *a $n - step$ cost function $\mathsf{cost_n}$,*
3. *a $n-$step abstract scheduler τ_n, and*
4. *a function $\mathbb{F}$,*

we can construct a $n-$step cost optimal monitor for the standard ordering $\leq$ of real numbers.

Note that a lot of the constraints of Thm. 6.1 have been relaxed. The lack of knowledge arbitrarily far in the future can actually help us to constraint and enumerate the variables that affect cost optimality and thus build an optimal monitor.

Although $n-$step cost-optimal monitors are practically relevant, they are limited in their capability to optimize the cost of the monitored system on a longer interval than they are supposed to. For example one cannot iterate an $n-$step cost-optimal monitor and expect to have the same results as using a (infinite-horizon) cost-optimal monitor.

7. Related Work

The first model of run-time monitors, *security automata*, was based on Büchi Automata [36]. Security automata observe individual executions of an untrusted application and halt the application if the execution is about to become invalid. Schneider observed that security automata enforce only safety properties, providing the first classification of policies enforceable by run-time monitors. Since then, several similar models have extended or refined the class of enforceable policies based on the enforcement and computational powers of monitors (e.g., [20,22,19,4]).

Recent work has revised these models or adopted alternate ones to more conveniently reason about applications, the interaction between applications and monitors, and enforcement in distributed systems. This includes Martinelli and Matteucci's model of run-time monitors based on CCS [30], Gay's et al. *service automata* based on CSP for enforcing security requirements in distributed systems [21], Basin's et al. language, based on CSP and Object-Z (OZ), for specifying security automata [5], and Mallios' et al. I/O automata-based model for reasoning about incomplete mediation and knowledge the monitor might have about the target [29].

Although these models are richer and orthogonal revisions to security automata and related computational and operational extensions, they maintain the same view of (enforceable) security policies: binary predicates over sets of executions. In this paper we take a richer view assigning costs and probabilities to traces and define cost-security policies and cost-enforcement, which as shown in §5 is a strict extension of binary-based security policies and enforcement.

Another difference is that some of these models (e.g., security and edit automata [36,25]) assume that the monitor has no information about the future behavior of targets. Other work has introduced models where monitors do have such information about the future behavior, e.g., through access to the source code of the target application (e.g., [22,29]). Our model introduces another source of information that the monitor could have about the future behavior of the target, i.e., expectation about its future behavior. For example, a security manger of a web-server that streams movies, can use data that has collected over

a number of days, to build a probabilistic model of future behavior of streaming clients. For instance, it could infer that during, or after, dinner time the requests for movies are really high. Then, if the cost of a DDOS monitor is prohibitive to run continuously, the security manager can make a decision to run the monitor only during rush-hours, and thus minimize its expected costs.

Note that by modeling the system in a probabilistic way, we consider averaging a mixed population of clients consisting of honest clients that behave correctly, honest clients that do not behave correctly by accident, and attackers that always try to violate the policy. In practice, the attacker will try to cause a huge cost by its own, but this cost will be balanced by well-behaved users. Hence, in this paper we try to focus on the expected cost of enforcement, rather than the worst case scenario, which assumes that all clients are actively attacking the system. In practical applications, focusing on the overall cost is often necessary: investing too much into expensive defenses for an unlikely attack can be as detrimental as a successful attack. (That said, our system still allows reasoning about the worst case, by selecting the scheduler and cost function accordingly.)

Drábik et al. introduce the notion of calculating the cost of an enforcement mechanism based on a relatively simple enforcement model, which does not include input/output actions or a detailed calculation of the execution probabilities [16]. To some extent, the notion of cost security policy defines a threshold characterising the maximal/minimal cost reachable, while taking the probability of reaching this threshold into account. Such a notion of threshold is also used by Cheng et al., where accesses are associated with a level of risk, and decisions are made according to some predefined risk thresholds, without detailing how such policies can be enforced at runtime [13]. In the context of runtime enforcement, Bielova and Massacci propose to apply a distance metrics to capture the similarity between traces, and we could consider the cost required to obtain one trace from another as a distance metrics [7].

An important aspect of this work is to consider that a property might not be locally respected, i.e., for a particular execution, as long as the property holds globally. This possibility is also considered, which quantifies the tradeoff correctness/transparency for non-safety boolean properties [15]. Caravagna et al. introduce the notion of lazy controllers, which use a probabilistic modelling of the system in order to minimize the number of times when a system must be controlled, without considering input/output interactions between the target and the environment as we do [11]. These lines of work are in the scope of going towards a notion of quantitative enforcement [31], where enforcement mechanisms are quantitatively evaluated and decisions made using quantitative analysis, rather than the binary adherence to a policy, in the context of process algebra.

Finally, the idea of optimal monitor is considered by Easwaran et al. in the context of software monitoring [17]. There, correcting actions are associated with rewards and penalties within a Directed Acyclic Graph, and the authors use dynamic programming to find the optimal solution. Similarly, Markov Decision Process (MDP) can be used to model access control systems [32], and the optimal policy can be derived by solving the corresponding optimisation problem. In our context, PIOA provide a more expressive framework to model enforcement scenarios, particularly due to the classification of actions to input, internal, and output. First, PIOA's state transitions do not have to satisfy the Markov property (i.e., the scheduler can discriminate based on the entire trace and not just the current state of the automaton) and thus they can model strictly more scenarios than MDPs. Second, the classification of actions to those that are under the automaton's control or those that are not, help us to better model the interaction between a target and a monitor, and the fact that there are target's decisions that are outside the monitor's control. Third, PIOA are equipped with a native notion of composition that helps with specifying the interactions between programs that we deal with in this paper. A potential lead for future work would therefore be to focus on Probabilistic I/O Automata that satisfy the Markov property and combine them with MDPs,

in order to reuse the computation of optimal policy of the latter within the expressive framework of the former.

Another direction for future work and constructing optimal monitors is to automate the synthesis of monitors by restricting the set of security policies to those that are expressible under a suitable logic. Previous work on run-time monitor synthesis has focused on variations of LTL [3,34,35] and shown how to construct a minimal monitor [6]. However, these approaches do not take into account probabilities, or generalized notions of cost (other than minimizing the monitor's size [6]). Previous work on synthesis, not for security purposes, that focuses on systems' synthesis from probabilistic components (e.g., [33]) or synthesis from specifications expressed in Probabilistic Computation Tree Logic (e.g., [2]) might be helpful.

8. Conclusion

We have introduced a formal framework based on probabilistic I/O automata to model and reason about interactive run-time monitors. In our framework we can formally reason about probabilistic knowledge monitors have about their environment and combine it with cost information to minimize the overall cost of the monitored system. We have used this framework to (1) calculate *expected costs of monitors* (§4), (2) define *cost security policies* and *cost enforcement*, richer notions of traditional definitions of security policies and enforcement [36] (§5), and (3) order monitors according to their expected cost and show how to build an optimal one (§6).

Acknowledgments

This work was supported in part by NSF grant CNS-0917047, by H2020 EU NeCS, by MIUR PRIN Security Horizons, and by the Army Research Laboratory under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

References

[1] A. Appel. Modern Compiler Implementation in ML: Basic Techniques. Cambridge University Press, 1997.

[2] C. Baier, M. Grober, M. Leucker, B. Bollig, and F. Ciesinski. Controller synthesis for probabilistic systems. In *In Proceedings of IFIP TCS*, 2004.

[3] H. Barringer, A. Goldberg, K. Havelund, and S. Koushik. Program monitoring with LTL in EAGLE. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, 2004.

[4] D. Basin, V. Jugé, F. Klaedtke, and E. Zălinescu. Enforceable security policies revisited. In *Proceedings of POST 2012*, volume 7215 of *Lecture Notes in Computer Science*, pages 309–328, 2012.

[5] D. Basin, E.-R. Olderog, and P. E. Sevinc. Specifying and analyzing security automata using CSP-OZ. In *Proceedings ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 70–81, 2007.

[6] A. Bauer, M. Leucker, and C. Schallhart. Runtime Verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20(4):1–64, 2011.

[7] N. Bielova and F. Massacci. Predictability of enforcement. In *Proceedings of the International Symposium on Engineering Secure Software and Systems*, pages 73–86, 2011.

[8] B. Blakley, E. McDermott, and D. Geer. Information Security is Information Risk Management. In *Proceedings of the 2001 Workshop on New Security Paradigms*, pages 97–104, 2001.

[9] R. Canetti, L. Cheung, D. Kaynar, M. Liskov, N. Lynch, O. Pereira, and R. Segala. Task-structured probabilistic I/O automata. Technical Report MIT-CSAIL-TR-2006-060, 2006.

[10] R. Canetti, L. Cheung, D. Kaynar, M. Liskov, N. Lynch, O. Pereira, and R. Segala. Task-structured probabilistic i/o automata. In *Proceedings of 8th International Workshop on Discrete Event Systems*, pages 207–214, 2006.

[11] G. Caravagna, G. Costa, and G. Pardini. Lazy security controllers. In *Proceedings of the 8th International Workshop on Security and Trust Management (STM 2012)*, pages 33–48, 2013.

[12] K. Chatterjee, L. Doyen, and T. A. Henzinger. Quantitative languages. In *Proceedings of the 17th International Conference on Computer Science Logic (CSL)*, pages 385–400, 2008.

[13] P.-C. Cheng, P. Rohatgi, C. Keser, P. A. Karger, G. M. Wagner, and A. S. Reninger. Fuzzy multi-level security: An experiment on quantified risk-adaptive access control. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, pages 222–230, 2007.

[14] M. R. Clarkson and F. B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6):1157–1210, September 2010.

[15] P. Drábik, F. Martinelli, and C. Morisset. A quantitative approach for inexact enforcement of security policies. In *Proceedings of the 15th international conference on Information Security*, ISC'12, pages 306–321, 2012.

[16] P. Drábik, F. Martinelli, and C. Morisset. Cost-aware runtime enforcement of security policies. In *Proceedings of the 8th International Workshop on Security and Trust Management (STM 2012)*, pages 1–16, 2013.

[17] A. Easwaran, S. Kannan, and I. Lee. Optimal control of software ensuring safety and functionality. Technical Report MS-CIS-05-20, University of Pennsylvania, 2005.

[18] U. Erlingsson. *The inlined reference monitor approach to security policy enforcement*. PhD thesis, Ithaca, NY, USA, 2004.

[19] Y. Falcone, J.-C. Fernandez, and L. Mounier. What can you verify and enforce at runtime? *Intl. Jrnl. Software Tools for Tech. Transfer (STTT)*, 14(3):349–382, June 2012.

[20] P. W. Fong. Access control by tracking shallow execution history. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, pages 43–55, 2004.

[21] R. Gay, H. Mantel, and B. Sprick. Service automata. In *Proceedings of the 8th international conference on Formal Aspects of Security and Trust*, pages 148–163, 2012.

[22] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *ACM Trans. Program. Lang. Syst.*, 28(1):175–205, January 2006.

[23] M. Kwiatkowska. Survey of fairness notions. *Information and Software Technology*, 31(7):371–386, September 1989.

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

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

[26] J. Ligatti and S. Reddy. A theory of runtime enforcement, with results. In *Computer Security - ESORICS 2010*, volume 6345 of *Lecture Notes in Computer Science*, pages 87–100, 2010.

[27] G. R. Malan, D. Watson, F. Jahanian, and P. Howell. Transport and application protocol scrubbing. In *Proceedings of INFOCOM 2000*, pages 1381–1390, 2000.

[28] Y. Mallios, L. Bauer, D. Kaynar, , and J. Ligatti. Enforcing more with less: Formalizing target-aware run-time monitors. Technical Report CMU-CyLab-12-009, CyLab, Carnegie Mellon University, 2012.

[29] Y. Mallios, L. Bauer, D. Kaynar, and J. Ligatti. Enforcing more with less: Formalizing target-aware run-time monitors. In *Proceedings of the 8th International Workshop on Security and Trust Management (STM 2012)*, pages 17–32, 2013.

[30] F. Martinelli and I. Matteucci. Through modeling to synthesis of security automata. *Electron. Notes Theor. Comput. Sci.*, 179:31–46, July 2007.

[31] F. Martinelli, I. Matteucci, and C. Morisset. From qualitative to quantitative enforcement of security policy. In *Proceedings of the 6th international conference on Mathematical Methods, Models and Architectures for Computer Network Security: computer network security*, MMM-ACNS'12, pages 22–35, 2012.

[32] F. Martinelli and C. Morisset. Quantitative access control with partially-observable markov decision processes. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, CODASPY '12, pages 169–180, 2012.

[33] S. Nain and M. Y. Vardi. Synthesizing Probabilistic Composers. In *Proceedings of the 15th International Conference on Foundations of Software Science and Computational Structures*, pages 421–436, 2012.

[34] G. Rocsu and S. Bensalem. Allen Linear (Interval) Temporal Logic – Translation to LTL and Monitor Synthesis. In *Proceedings of the 18th International Conference on Computer Aided Verification*, pages 263–277, 2006.

[35] G. Rocsu, F. Chen, and T. Ball. Synthesizing Monitors for Safety Properties: This Time with Calls and Returns. In *Proceedings of Runtime Verification*, pages 51–68, 2008.

[36] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3:30–50, February 2000.

[37] N. Xie, N. Mead, P. Chen, M. Dean, L. Lopez, D. Ojoko-Adams, and H. Osman. SQUARE Project: Cost/Benefit Analysis Framework for Information Security Improvement Projects in Small Companies. Technical Report CMU/SEI-2004-TN-

045, Software Engineering Institute, Carnegie Mellon University, 2004.

Appendix 1. Proofs of theorems

Proposition 6.1. *Given a system S there is a cost optimal monitor M (for some well-order $\trianglelefteq$).*

Proof sketch. There are countably many monitors (since there are countably many I/O automata). Thus, there are countably many monitored systems (for the given system S). Moreover, it is known (from set theory) that every countable set can be well-ordered by some well-order $\trianglelefteq$. By definition of well-orders every subset of monitored systems will have a least element under $\trianglelefteq$. Thus, the least element $M \times S$ of the set of all possible monitors for S exists, and M is cost optimal for S.

□

Theorem 6.1. *Given:*

1. *a finite-state system S,*
2. *a cost assignment* map *of each action of S to a real value such that for every input action i of* $\mathsf{map}(i) = 0$, ,
3. *a cost function* cost *that is defined recursively based on* map *such that the* cost *of a trace t is the sum of the values of the actions that appear on t,*
4. *a weakly fair*[14] *abstract scheduler τ, and*
5. *a function $\mathbb{F}$ that is monotone and continuous (i.e., it preserves limits),*

we can construct a cost optimal monitor for the standard ordering $\leq$ of real numbers.

Proof sketch. First we define the signature of M to contain (1) as input actions the output actions of S, and (2) as output actions the input actions of S. Since S has finitely many states we can detect (1) whether it contains any cycles, and (2) the cost of each cycle. Assuming that S contains cycles, we construct a monitor that “complements” every positive-value cycle k of S, i.e., the transition relation of M produces all possible cycles k that have a positive sum of costs. Since M and S contain complementary input and output actions, the composition $M \times S$ will exhibit all those cycles.

Since the abstract scheduler is weakly fair, it is easy to see that this will also hold for the refined scheduler σ for $(M \times S)$ (by definition of refinement). This guarantees that $(M \times S)$ will contain infinite traces. In fact, $M \times S$ will contain all possible infinite traces that S can produce and whose costs will diverge to $+\infty$.

Finally, every other monitor M' will either (1) contain less cycles with positive cost than M, and thus will diverge (if they do) “slower”, or (2) contain negative cycles in addition to the positive ones which will have the same effect as the previous point, or (3) may contain additional self loops of output actions since they are under the monitor’s control: by the second and third constraint of the theorem the only actions that have a cost are the outputs of the system, i.e., the outputs of any M' will have a zero impact on the cost of a trace. Thus, in conclusion, by the monotonicity of $\mathbb{F}$, M will be optimal as compared to every other M'.

□

[14]As described in §5, weakly fair schedulers are the schedulers that do not assign zero probabilities to steps of an automaton from a given state (if such steps are defined in the transition relation of the automaton).

Theorem 6.2. *Given a number $n \in \mathbb{N}$ and:*

1. *a finite-state system S,*
2. *a $n - step$ cost function $\mathsf{cost_n}$,*
3. *a $n-$step abstract scheduler τ_n, and*
4. *a function $\mathbb{F}$,*

we can construct a $n-$step cost optimal monitor for the standard ordering $\leq$ of real numbers.

Proof sketch. One can, inefficiently, enumerate all possible values for traces that the system can exhibit for $n-$steps and find the actions that the monitor needs to exhibit in order to be cost optimal. Since the traces that the monitors needs to exhibit to be $n-$step optimal are finite (both in length and cardinality), one can easily build a monitor that exhibits those (finitely many) traces by interleaving fresh states amongst each of those traces.

□