Probabilistic Cost Enforcement of Security Policies

Yannis Mallios[1], Lujo Bauer[1], Dilsun Kaynar[1],
Fabio Martinelli[2], and Charles Morisset[3]

[1] Carnegie Mellon University, Pittsburgh, PA, USA
[2] Istituto di Informatica e Telematica, National Research Council, Pisa, Italy
[3] Newcastle University, Newcastle, UK

Abstract. This paper presents a formal framework for run-time enforcement mechanisms, or monitors, based on probabilistic input/output automata [3, 4], 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 [17], 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.

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.

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. Such guarantees can be achieved through the use of formal reasoning.

Schneider introduced security automata [22], 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., [17]). 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 [18]. 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 [18]) depicts the differences between the two mechanisms in a specific scenario. Although both mechanisms 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).

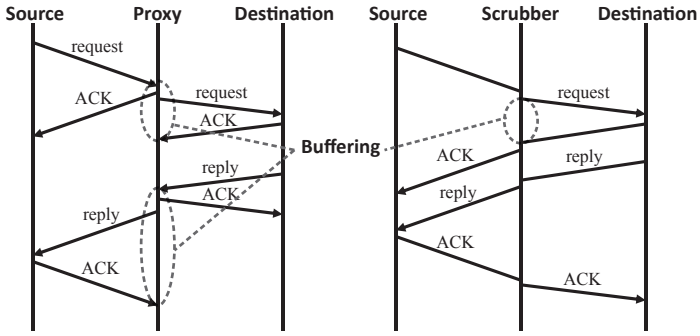


Fig. 1: TCP transport layer proxies and scrubbers. The circled portions represent the amount of time that data is buffered.

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 [10]; 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 [19]. 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. Finally, one might also wish to ensure that a monitor enforces a cost policy, which defines which costs are acceptable; practical cost policies can depend on a probabilistic model of the system's behavior, e.g., take into account the likelihood of particular events. 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.

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 [3, 4]. 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 [22]. 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 Background

We introduce our notation in §2.1 and then briefly review probabilistic I/O automata (PIOA) [3, 4] in §2.2; more details can be found in our technical report [20] or standard PIOA references, e.g., [3, 4]. In §2.3 we extend PIOA by introducing the notion of *abstract schedulers*, which we use in the cost comparison of monitors in §5. Finally, in §2.4, 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.

2.1 Preliminaries

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 $\mu(X) \leq 1$. We use $\mathsf{Disc}(X)$ and $\mathsf{SubDisc}(X)$ to denote, respectively, the set of discrete probability measures and discrete sub-probability measures on X. If μ is a probability measure then use $\mathsf{supp}(\mu)$ to denote the set of elements that have non-zero measure.We let $\delta(x)$ denote 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 we denote by $\mu_1 \times \mu_2$ the *product measure*, such that $\mu_1 \times \mu_2(x, y) = \mu_1(x) \cdot \mu_2(y)$ (i.e., component-wise multiplication).

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 $\mathcal{F}_X$. 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 $f(\mu)(Y) = \mu(X)$.

2.2 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 environment. 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[4] 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 $\mathsf{trace}(e)$, 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 $\mathsf{trace}(e) = t$. We use $\mathsf{execs}(P)$ and $\mathsf{traces}(P)$ (resp., $\mathsf{execs}^*(P)$ and $\mathsf{traces}^*(P)$) 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, $\sigma_1 \prec \sigma_2$.

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. When the signatures are compatible we say that the corresponding automata 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. $sig(P) = \prod_{i \in I} sig(P_i) = \Big(output(P) = \cup_{i \in I} output(P_i),\quad internal(P) = \cup_{i \in I} internal(P_i),\quad input(P) = \cup_{i \in I} input(P_i) - \cup_{j \in I} output(P_j)\Big)$;
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 $q_i \in q$, $i \in I$ and
   (b) for all $i \in I$ if $a \in acts(P_i)$ then $(q_i, a, \mu_i) \in R(P_i)$, otherwise $\mu_i = \delta(q_i)$.

Nondeterministic choices in P are resolved using a *scheduler*. A *scheduler* for P is a function $\sigma : \mathsf{execs}^*(P) \to \mathsf{SubDisc}(R(P))$ s.t., if $(q, a, \mu) \in \mathsf{supp}(\sigma(e))$ then $q =$

[4] If a PIOA P has a transition $(q, \alpha, \mu) \in R(P)$ then we say that action α is *enabled* in state 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 [3, 4]. 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.

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.

2.3 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 (§5). 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[5] 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 $t''; a$, $t \preceq t''$.

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

[5] 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.

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 $t \notin \mathsf{traces}(P)$ or $\tau(t')(\{a\}) = 0$, 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 $A = \{x \in extern(S) \mid t'; x \in \mathsf{traces}(P)\}$.

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 $\mathcal{F}_{P_T}$.

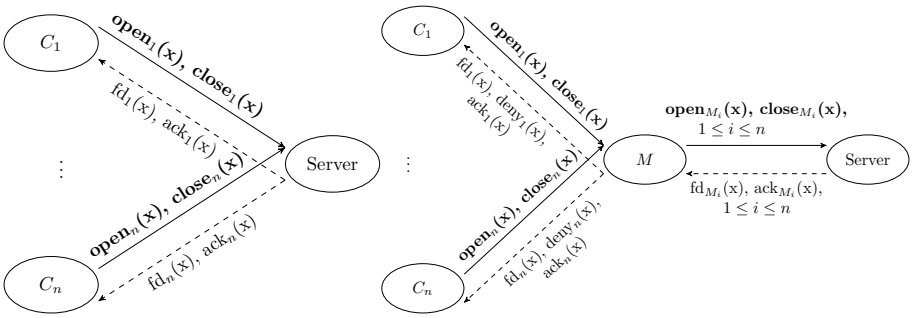
We now formalize the relationship between schedulers and abstract schedulers. Given an abstract scheduler τ over a signature S, and a PIOA P with $sig(P) = S$, a scheduler σ is *derivable* from τ iff σ is a scheduler for P such that for all executions $e \in \mathsf{execs}(P)$ the trace distributions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

2.4 Running Example Modeled Using PIOA

To illustrate how our framework can be used to model enforcement scenarios we will consider a running example of a file server S, illustrated in Fig. 2a.

Clients (C_1 through C_n in the figure) can request to open or close a particular file. The server responds to the requests by returning a file descriptor or an acknowledgment that the file was closed successfully. Given a security policy P stating that at most one client at a time can access a particular file, a monitor is interposed between the clients and the server to enforce P (Fig. 2b). The monitor has the ability to *deny* access to a file requested by a client.

We now show how to model the running example using PIOA. Each client C_i requests to open a file x through an $open_i(x)$ output action. Once the client receives a file descriptor through an $fd_i(x)$ input action, it requests to close the file through an $close_i(x)$ action. When it receives an acknowledgment that the file was closed, it stops



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

Fig. 2: Diagrams of interposing a monitor between clients and server

requesting access to the file. If, however, the client is denied access to the file, it probabilistically chooses between requesting the file again and permanently discontinuing requesting the file.

A state diagram of C_i is shown in Fig. 3.[6] 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.

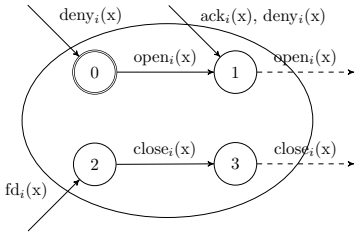


Fig. 3: Client PIOA state transition diagram

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

[6] Pseudocode and additional state diagrams for clients and the server can be found in our technical report [20].

Signature: Input: $open_i(x)$,$close_i(x)$, $fd_{M_i}(x)$, $ack_{M_i}(x)$, where x is a filename
Output: $open_{M_i}(x)$, $close_{M_i}(x)$, $fd_i(x)$, $ack_i(x)$, $deny_i(x)$, where x is a filename

States: p: list (of triples) of requests from clients to monitor
q: list (of triples) of responses from monitor to clients
r: list (of pairs) of active connections

Start States: $p = q = r = nil$

Transitions:

$open_i(x)$
Effect: $p := p@[\langle op, i, x\rangle]$

$close_i(x)$
Effect: $p := p@[\langle cl, i, x\rangle]$

$fd_{M_i}(x)$
Effect: $q := q@[\langle fd, M_i, x\rangle]$

$ack_{M_i}(x)$
Effect: $q := q@[\langle ack, M_i, x\rangle]$

$open_{M_i}(x)$
Precondition: $p = \langle op, i, x\rangle :: p'$
$\wedge\ \nexists\langle x, j\rangle \in r, j \neq i$
Effect: $p := p'$
$r := r@[\langle x, i\rangle]$

$close_{M_i}(x)$
Precondition: $p = \langle cl, i, x\rangle :: p'$
Effect: $p := p'$
$r := r\backslash[\langle x, i\rangle]$

$fd_i(x)$
Precondition: $q = \langle fd, M_i, x\rangle :: q'$
Effect: $q := q'$

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

$deny_i(x)$
Precondition: $p = \langle op, i, x\rangle :: p'$
$\wedge\exists\langle x, j\rangle \in r, j \neq i$
Effect: $p := p'$

Fig. 4: M_{DENY} PIOA definition

The pseudocode[7] for M_{DENY} is depicted in Fig. 4. The pseudocode for M_{PROB} is similar and can be found in our technical report [20], along with additional details about the structure of the monitors.

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 $q_\Pi = \langle q_{C_1}, \ldots, q_{C_n}, q_M, q_S\rangle$. An example execution for M_{DENY} is: $e_{M_{DENY}} = q_{\Pi_0}\ open_1(x)\ q_{\Pi_1}\ open_{M_1}(x)\ q_{\Pi_2}\ fd_{M_1}(x)\ q_{\Pi_3}\ fd_1(x)\ q_{\Pi_4}\ open_2(x)\ q_{\Pi_5}\ deny_2(x)\ q_{\Pi_6}\ open_2(x)\ q_{\Pi_7}\ deny_2(x)\ q_{\Pi_8}$. The trace of $e_{M_{DENY}}$ is: $t_{M_{DENY}} = \mathsf{trace}(e_{M_{DENY}}) = open_1(x)\ open_{M_1}(x)\ fd_{M_1}(x)\ fd_1(x)\ open_2(x)\ deny_2(x)\ open_2(x)\ deny_2(x)$.

In $t_{M_{DENY}}$ client C_1 asks to open file x and is given access, after which client C_2 asks to open the same file and is denied access by the monitor.

Let us consider the scheduler σ that schedules transitions based on the following high-level pattern: $\Big([C_1, \ldots, C_n];\ M^*;\ S;\ M^*\Big)^\infty$. This pattern says that σ chooses equiprobably 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 $\epsilon_{\sigma,\bar{q}}(C_{e_{M_{DENY}}}) = \frac{0.1}{n^2}$. Similarly, we can calculate the probabilities of $t_{M_{DENY}}$ (more details can be found in our technical report [20]).

[7] We use the precondition pseudocode style that is typical in I/O automata papers (e.g., [3, 4]).

3 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 trace over a signature S, i.e., every possible sequence of external actions of S. More formally, a *cost function* is a signed measure cost on the $\sigma-$field $\mathcal{F}_{P_T}$ generated by cones of traces, i.e., $\mathsf{cost} : \mathcal{F}_{P_T} \rightarrow [-\infty, \infty]$, where P_T are the traces of an automaton P with signature S that generates all possible traces of its signature. 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 they infer. Although traces are the subject of our analysis, cones are their (sound) mathematical representation.

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

Given a scheduler σ and a cost function cost, the *probabilistic cost of a PIOA* P is defined as $\mathsf{pcost}^{\mathbb{F}}_\sigma(P) = \mathbb{F}_{t\in\mathsf{traces}(P)}(\mathsf{pcost}_\sigma(C_t))$. 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. [6]): (1) $\mathsf{Sup}(v) = sup\{v_n \mid n \geq 0\}$, or (2) $\mathsf{LimInf}(v) = liminf_{n\rightarrow\infty} v_n = lim_{n\rightarrow\infty} inf\{v_i \mid i \geq n\}$. Sup chooses the maximal number that appears in v (e.g., the maximal temperature that a system can reach). LimInf chooses the minimal number that appears infinitely often in v (e.g., the temperature that the system goes down to infinitely often).

If $\mathsf{cost}_\sigma(C_t) \geq 0$ for some trace t, then we call $\mathsf{cost}(C_t)$ the *value of* t. If $\mathsf{cost}(C_t) \leq 0$, then the absolute value of $\mathsf{cost}_\sigma(C_t)$ is the *cost of* t. We define similarly the probabilistic value and cost of a trace t and a PIOA P.

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

In our technical report we show how one can define the cost of a system given cost functions for its components [20]: such an approach can be used to embed the framework of Drabik et al. [10] in ours, showing that our framework is at least as expressive.

4 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 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.[8]

The cost function defined in §3 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 open 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.

Given a (monitored) system P, a *cost security policy* over $sig(P)$ is a cost function, i.e., a signed measure Pol on the $\sigma-$field $\mathcal{F}_{P_T}$ generated by cones of traces that range over $sig(P)$, i.e., $\mathsf{Pol} : \mathcal{F}_{P_T} \rightarrow [-\infty, \infty]$. 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 [22]: 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., [22, 8, 19]) for future work.

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. Given a scheduler σ, a cost function cost, a policy Pol, 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 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.,:

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

[8] By assumption, M and S are compatible. In scenarios where this is not the case, one can use renaming to make the automata compatible [19, 3, 4].

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

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 (resp. greater) than or equal to 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 §3, this value could represent the maximum value 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 traces, 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., [22]) 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 $0-\text{enforces}_{\leq}$ 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 [17]: 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 $0-\text{enforcement}_{\leq}$ 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 $0-\text{enforces}_{\leq}$ the cost security policy Pol_P under $\mathbb{F}_P$ and $\mathsf{cost}(\_) = 0$.

A notion of transparency is often used to define practically useful policy enforcement (e.g., [17]). Due to space constraints, we discuss this in our technical report [20].

5 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 §2.3. Namely, to compare two monitored systems we use a single abstract scheduler which we then refine into schedulers for each monitored system.[9]

[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

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*. 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) the monitors get a chance to respond to targets' actions infinitely often (i.e., the monitors are not starved), 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 $a \in extern(M)$, 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. 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 [16]) and non-zero probability to monitors that output "valid" target actions verbatim (i.e., the scheduler allows suppression monitors [16]). 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. There might be scenarios where such schedulers are appropriate[10], but in this paper we pursue only the equiprobable scenario.

Given a system S, a function $\mathbb{F}$, a function pcost, two monitors M_1 and M_2 with $sig(M_1) = sig(M_2)$, an abstract scheduler τ over $sig(M_1 \times S)$, and two schedulers σ_1 (for $M_1 \times S$) and σ_2 (for $M_2 \times S$) derivable from τ, we say that M_2 is less costly than a monitor M_1 and write $M_2 \leq M_1$, if and only if $\mathsf{pcost}^{\mathbb{F}}_{\sigma_2}(M_2 \times S) \leq \mathsf{pcost}^{\mathbb{F}}_{\sigma_1}(M_1 \times S)$. Note that in the particular case where $\mathsf{pcost}^{\mathbb{F}}_{\sigma}$ corresponds to the expected cost of all the traces in $M \times S$, the ordering relation $\leq$ roughly corresponds to the notion of "globally more-efficient" of [10]. A monitor M is *cost optimal* for a system S if and only if for all monitors M' with $sig(M) = sig(M')$, $M \leq M'$.

The next theorem formalizes the intuition that a monitor that exploits knowledge about the scheduler and the cost function should be more cost efficient than monitors that do not. The theorem shows that such knowledge can be exploited to build a cost optimal monitor. Note that in the theorem the cost function and scheduler are universally quantified, i.e., the monitor is cost optimal for any abstract scheduler and cost function.

Theorem 1. *Given an abstract scheduler τ and a function $\mathbb{F}$ that is monotone*[11] *and continuous (i.e., it preserves limits), there is a cost-optimal monitor that optimizes its transitions based on a scheduler σ (derived from τ) and cost function* cost[12].

Thm. 1 provides a generic description of the conditions sufficient for constructing a cost-optimal monitor. In the constructive proof of Thm. 1 we build a monitor that keeps

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 [16], where the insertion monitor might exhibit additional actions, e.g., logging.

[10] This is a similar situation with having various definitions for fairness [15].

[11] Given two sets of real numbers $X, Y \in 2^{\mathbb{R}}$ we write $X \sqsubseteq Y$ if and only $\forall x \in X : \exists y \in Y : x \leq y$. We write $x \sqsubseteq y$ for $\{x\} \sqsubseteq \{y\}$, i.e., $x \sqsubseteq y \Leftrightarrow x \leq y$. We say that a function $f : 2^{\mathbb{R}} \to \mathbb{R}$ that is *monotone* if and only if it is monotone under the ordering $\sqsubseteq$, i.e., if $X \sqsubseteq Y$ then $f(X) \sqsubseteq f(Y)$.

[12] Proofs can be found in our technical report [20].

at its state the past execution, and at each state the next transition taken by the monitor minimizes the expected cost of the trace using σ and cost in its calculation.

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, in the running example, one way monitors can modify the behavior of the system is by denying an access to a client. If we 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 §2.4. 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 $\mathsf{sup}_{t \in \mathsf{traces}(\Pi_{M_{DENY}})}(\mathsf{pcost}_{\sigma_{M_{DENY}}}(C_t))$, and similarly for M_{PROB}.

We first observe that with such a cost function, the maximal (i.e., best) reachable cost is 0, meaning that no deny action is 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.

Second, 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 $p_1 < p_2$, 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 $M_{PROB} \leq M_{DENY}$.

Finally, 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 Related Work

The first model of run-time monitors, *security automata*, was based on Büchi Automata [22]. Security automata observe individual executions of an untrusted application and halt the application if the execution is about to become invalid. 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., [12, 14, 11]).

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 [21], Gay et al.'s *service automata* based on CSP for enforcing security requirements in distributed systems [13], Basin et al.'s language, based on CSP and Object-Z (OZ), for specifying security automata [1], and Mallios et al.'s I/O automata-based model for reasoning about incomplete mediation and knowledge the monitor might have about the target [19]. 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 §4, is a strict extension of binary-based security policies and enforcement.

Drábik et al. introduce the notion of calculating the cost of an enforcement mechanism [10], based on a relatively simple enforcement model that does not include input/output actions or a detailed calculation of the execution probabilities. To some extent, the notion of cost security policy defines a threshold characterizing 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 [7]. In the context of runtime enforcement, Bielova and Massacci propose to apply a distance metrics to capture the similarity between traces [2], and we could consider the cost required to obtain one trace from another as a distance metrics.

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 by Drabik et al., who quantify the tradeoff correctness/transparency for non-safety boolean properties [9]. Caravagna et al. introduce the notion of lazy controllers, which use a probabilistic modeling 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 [5].

7 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* (§3), (2) define *cost security policies* and *cost enforcement*, richer notions of traditional definitions of security policies and enforcement [22] (§4), and (3) order monitors according to their expected cost and show how to build an optimal one (§5).

Acknowledgments This work was supported in part by NSF grant CNS-0917047 and by EU FP7 projects NESSoS and SESAMO.

References

1. 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.
2. N. Bielova and F. Massacci. Predictability of enforcement. In *Proceedings of the International Symposium on Engineering Secure Software and Systems*, pages 73–86, 2011.

3. 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.
4. 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.
5. 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.
6. 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.
7. 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.
8. M. R. Clarkson and F. B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6):1157–1210, September 2010.
9. 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.
10. 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.
11. 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.
12. 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.
13. 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.
14. 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.
15. M. Kwiatkowska. Survey of fairness notions. *Information and Software Technology*, 31(7):371–386, September 1989.
16. 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.
17. 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.
18. G. R. Malan, D. Watson, F. Jahanian, and P. Howell. Transport and application protocol scrubbing. In *Proceedings of INFOCOM 2000*, pages 1381–1390, 2000.
19. 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.
20. Y. Mallios, L. Bauer, D. Kaynar, F. Martinelli, and C. Morisset. Probabilistic cost enforcement of security policies. Technical Report CMU-CyLab-13-006, CyLab, Carnegie Mellon University, 2013.
21. F. Martinelli and I. Matteucci. Through modeling to synthesis of security automata. *Electron. Notes Theor. Comput. Sci.*, 179:31–46, July 2007.
22. F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3:30–50, February 2000.