# A Modular Runtime Enforcement Model using Multi-Traces

Rania Taleb, Sylvain Hallé, and Raphaël Khoury

Laboratoire d'informatique formelle
Université du Québec à Chicoutimi, Canada

**Abstract.** Runtime enforcement seeks to provide a valid replacement to any misbehaving sequence of events of a running system so that the correct sequence complies with a user-defined security policy. However, depending on the capabilities of the enforcement mechanism, multiple possible replacement sequences may be available, and the current literature is silent on the question of how to choose the optimal one. In this paper, we propose a new model of enforcement monitors, that allows the comparison between multiple alternative corrective enforcement actions and the selection of the optimal one, with respect to an objective user-defined gradation, separate from the security policy. These concepts are implemented using the event stream processor BeepBeep and a use case is presented. Experimental evaluation shows that our proposed framework can dynamically select enforcement actions at runtime, without the need to manually define an enforcement monitor.

## 1 Introduction

Runtime enforcement is the process of monitoring a program during its execution, and intervening as needed to ensure compliance with a user-specified security policy [13]. The process differs from runtime *verification* in that the monitor is expected to provide a valid replacement for any misbehaving trace, rather than simply signal a violation. In recent years, the growing presence of smart contracts has sparked a renewed interest in runtime enforcement [8]; indeed, since smart contracts cannot be modified after deployment, runtime enforcement is the only remedial mechanism available to handle a deviation from the expected behavior.

The notion of enforcement is commonly defined in terms of two properties: *soundness* and *transparency* [19, 25]. Soundness imposes that the output of the monitor must respect the underlying security policy; on its side, transparency states that if the original security policy was already valid, then the replacement sequence must be equivalent, with respect to some equivalence relation. In addition, Khoury *et al.* suggested that the transformations performed on invalid traces also be bounded by an equivalence relation or a preorder [22]. Indeed, one would rarely accept a security enforcement mechanism that corrects an invalid trace by replacing it with a completely unrelated valid trace. This line of research has also stressed the need for a process that selects the optimal corrective action, from a set of possible alternatives that are available to the monitor.

Many formal models of enforcement have been proposed in the past to capture the behavior of monitors [2, 19, 20, 23, 25]. In most of them, a single mathematical structure (usually an automaton or some other type of finite state machine) is tasked with the entirety of the enforcement process: reading the input, transforming it through a process of substitutions, insertions, deletions and/or truncations, and ensuring compliance of the resulting output trace with respect to both soundness and transparency. As a consequence, elaborate proofs are often required to ensure that the output of the monitor is indeed sound and transparent (see e.g. [22]).

The present paper offers a different take on the problem, and introduces a model of runtime enforcement composed of three separate stages. The first stage transforms events of an (invalid) input trace into a set of traces, obtained by applying each possible modification one is allowed to apply. The second stage filters this set to keep only the traces that do not violate a specified security policy, while the third stage ranks the remaining traces based on an objective gradation we term the *enforcement preorder*, and picks the highest-scoring trace as its output.

This design provides a high level of modularity. First, the expression of the allowed modifications to the trace, the security policy itself and the enforcement preorder can all be expressed independently, using a different formal notation if need be. Second, the model does not require a specific enforcement monitor to be manually synthesized for each policy to enforce: corrective actions are computed, selected and applied dynamically. Finally, the model does not impose a single valid output, and rather allows multiple corrective actions to be compared against the enforcement preorder provided by the user.

The remainder of this paper is organized as follows: Section 2 provides a more detailed statement of the problem this paper seeks to address, while simultaneously reviewing previous work on the topic. Then, in Section 3, we present a new model of monitors for runtime enforcement. Section 4 illustrates the flexibility of the approach with a use case adapted from the literature, and presents a concrete implementation of the principle as an extension of the BeepBeep event stream processing library [18]. Concluding remarks are given in Section 5.

## 2 State of the Art in Runtime Enforcement

Let $\Sigma$ be a finite or countably infinite set of elements called *events*. The set of all finite sequences from $\Sigma$, also called *traces*, is given as $\Sigma^*$. Given a trace $\sigma \in \Sigma^*$, we use the notation $\sigma[i]$ to range over the elements of $\sigma$, where $i$ represents the event at the $i$-th position (the first event is at $i = 0$). The notation $\sigma[i..]$ denotes the remainder of the sequence starting from action $\sigma[i]$ while $\sigma[..i]$ denotes the prefix of $\sigma$, up to its $i$-th position. The concatenation of two sequences $\sigma$ and $\sigma'$ is given as $\sigma \cdot \sigma'$. The empty sequence is denoted $\epsilon$, and $\sigma \cdot \epsilon = \epsilon \cdot \sigma = \sigma$. As usual, the notation $\sigma' \preceq \sigma$ denotes that $\sigma'$ is a prefix of $\sigma$.

Given two sets of events $\Sigma_1$, and $\Sigma_2$, a trace transducer is a function $\tau : \Sigma_1^* \to \Sigma_2^*$, with the added condition that for every $\sigma, \sigma' \in \Sigma^*$, $\sigma' \preceq \tau(\sigma)$ implies $\sigma' \preceq \tau(\sigma \cdot x)$ for every $x \in \Sigma_1$. In other words, a transducer takes as input a sequence of events, and progressively outputs another sequence of events. Given an arbitrary transducer $\tau : \Sigma_1^* \to \Sigma_2^*$ and a sequence $\sigma \in \Sigma_1^*$, we define $\tau_\sigma : \Sigma_1^* \to \Sigma_2^*$ as $\tau_\sigma(\sigma'') = \tau(\sigma \cdot \sigma'')$. Intuitively, $\tau_\sigma$ is a device abstracting the "internal state" of the transducer $\tau$ after ingesting the events from the prefix $\sigma$.

Of particular importance in this paper is an enforcement monitor, which is defined as a transducer $\tau : \Sigma^* \to \Sigma^*$. A security policy is a subset $S \subseteq \Sigma^*$ of sequences called the *valid* sequences. An enforcement monitor is said to satisfy the *soundness* condition if $\tau(\sigma) \in S^*$ for every $\sigma \in \Sigma^*$. Additionally, depending on the type of monitor used, it may be subject to other constraints that limit the freedom of the monitor to substitute one sequence for another (a property we call *transparency*).

A long line of research focuses on delineating the set of properties that are (or are not) enforceable by monitors operating under a variety of constraints [2, 22, 25]. A key finding of these works is that the enforcement power of monitors is affected both by the capabilities of the monitor as an enforcement mechanism, and by the license given to the monitor to alter the input sequence (the transparency requirement). A thorough survey of runtime enforcement, stressing its connection to runtime verification, is given by Falcone *et al.* [13].

## 2.1 Monitor Capabilities

In his initial formulation, Schneider [25] considered a monitor that observes the sequence of events produced by the target program, and reacts by aborting the execution (truncating the execution sequence) upon encountering an event which, if appending that event to the ongoing execution, would violate the security policy. Ligatti *et al.* [2] consider more varied models of monitors, capable of inserting events in the execution stream, of suppressing the occurrence of some events while allowing the remainder of the execution to proceed, or both. Another characterization, in which some events lie beyond the control of the monitor, was proposed by Khoury *et al.* [20].

Extending the available capabilities given to a monitor to alter the input trace greatly extend its enforcement power, but may in counterpart introduce several possible corrective courses of action to restore compliance with a policy. For instance, a trace where a *send* action occurs immediately after a file is being *read* violates a policy stipulating that no information can be sent on the network after reading from a secret file, unless the sending is recorded in a log beforehand. Multiple corrective actions are hence possible: aborting the execution before the *send* action (truncation); inserting an entry in the log (insertion); or suppressing either the *read* or the *send* action (suppression).

In this line of research, the monitor is usually modeled as a finite state machine, which dictates its behavior according to the input action and its current state. Care must be taken to ensure that this FSM correctly enforces the policy and is concordant with the limitations imposed on the monitor's capabilities. Falcone *et al.* [14] showed that a finer automaton model, with explicit store and dump operations, can enforce policies in the *response* class from the safety-progress classification [6]. Their model also lends itself to implementation in a more straightforward manner than previous models.

Another line of research examines how memory constraints affect the enforcement power of monitors. Thali *et al.* [28] study the enforcement power of monitors with bounded memory; Fong *et al.* [15] study a monitor that only records the shallow history (i.e. the unordered set of events) of the execution, while Beauquier *et al.* [3] study the enforcement power of a monitor with finite, but unbounded memory. On their side, the monitors proposed by Ligatti *et al.* and Bielova *et al.* have the capacity to store an unbounded quantity of program events, simulating the execution until it can ascertain that the ongoing execution is valid; however, this course of action may not always be possible in practice. In contrast, Dolzhenko *et al.* propose a model of monitoring in which the monitor is required to react to each action performed by the target program as it occurs [10].

## 2.2 Transparency Constraints

In the original definition of runtime enforcement reported above, the notion of transparency only imposes that the monitor must maintain the semantics of *valid* sequences [2], which can lead to undesirable behavior. As an example, consider the policy “an opened file is eventually closed”, and a sequence in which multiple files are consecutively opened and closed, except the final file which is opened, but not closed. The monitor may correct the situation either by appending a close action at the end of the sequence, or by deleting the opening of the ultimate file and any subsequent file actions (reads and writes). However, the monitor could also enforce the property by removing every well-formed pair of files being opened and closed, or even by adding to the sequence new events not present in the original. This is because the definition of enforcement entails that the monitor can replace an invalid sequence with *any* valid sequence, even one completely unrelated to the original execution.

*Transparency* constraints refer to mechanisms by which the available enforcement actions of a monitor are restricted according to some requirement. For example, Bielova *et al.* create sub-classes that further constrain the monitor’s handling of invalid executions [5]. First is the class of monitors that are limited to delaying the execution of some program events, but may not insert new events into the execution; second, monitors that may only insert the delayed part of the execution on an all-or-nothing basis; third, monitors limited to output some prefix of invalid sequences. They compare the set of properties that are enforceable in each case.

Khoury *et al.* also consider constraints on invalid sequences, and introduce the notion of “gradation” of solutions [22]. Sequences are arranged on a partial order, independent of the security policy being enforced, which makes it possible to state that some corrective actions are preferable to others. For example, a policy stating that every acquired resource must eventually be relinquished could be enforced by forcibly removing the resource from the control of a principal and reallocating it to another user; a monitor could then seeking to allocate the resource equitably between all users, or to minimize the amount of time the resource is idle. In a similar vein, Drábik *et al.* [12] propose to associate each action taken by the monitor with a cost, and to seek optimal cost. Their notion of transparency binds the monitor in its handling of both valid and invalid sequences; it is defined as a function $f : \Sigma^* \rightarrow \mathbb{R}$, which the monitor must either maximize or minimize, depending on its formulation. This is the work that is most closely related to the current study.

A few elements stand out in this line of research. First, most approaches impose on the designer to create a finite state machine that enforces the desired policy, and respect any limitations on the capabilities of the monitor (with the exception of [14], which provides a monitor synthesis algorithm). This is a non-trivial task, made even harder when some guarantee of optimal enforcement cost is sought. Furthermore, elaborate proofs are often required to ensure that the enforcement of the property is correct, transparent and optimal.

The use of a fixed cost for each program action is limiting. One may prefer a more flexible gradation of solutions, in which the value associated with a solution is more context-specific.

## 3 A Modular Runtime Enforcement Pipeline

In this paper, we present an alternate model of runtime enforcement with the aim to transform the input sequence in order to ensure both the respect of the security policy as well as provide assurance that the corrected sequence is optimal with respect to a separate transparency requirement. The key idea of this model is to separate the various operations of enforcement into independent computation steps. The high-level schematics of the model are illustrated in Figure 1. Various transducers are represented as boxes illustrated with different pictograms, depending on their definition. These transducers are organized along a "pipeline" where events flow from left to right. A link between two transducers indicates that the output of the first is given as the input to the second.

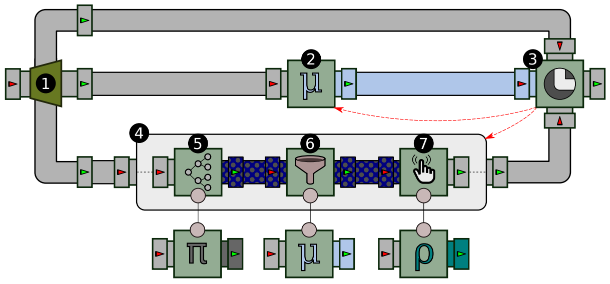


Fig. 1: The stages of the runtime enforcement framework.

An input event sequence is first forked into three separate copies, as represented by box #1 in the figure. One copy is fed to an instance of a transducer $\mu$ called the *monitor* (box #2), which determines whether the input trace violates the security policy. Another copy is fed to an enforcement pipeline (box #4), itself decomposed into three phases. First, a single event sequence is turned into multiple event sequences by applying the possible corrective actions produced by a proxy transducer $\pi$ (#5); this set of sequences is then filtered out so that only sequences satisfying the security policy evaluated by $\mu$ are kept (#6). The last phase sends the remaining sequences into a ranking transducer $\rho$, and picks the one with the highest rank as specified by the enforcement preorder (#7).

The last step of the pipeline is represented by box #3, which is called a *gate*. Based on the output from the monitor (box #2), the gate either outputs elements of the original trace directly (if it is valid), or switches to the output from the enforcement pipeline emitting a corrected sequence. Depending on the actual sequence of events produced by the gate, the internal state of the upstream transducers may need to be forcibly updated; this process, called *checkpointing*, is represented by the backwards red arrows. In the remainder of this section, we describe the stages of this pipeline in more detail.

## 3.1 Production of Corrected Traces

As discussed earlier, an enforcement monitor can apply a combination of several modifications to an input trace. These possible alteration actions are encapsulated into a conceptual entity that we call a proxy $\pi$, corresponding to the first stage of the enforcement pipeline represented by box #4.

Formally, the proxy is a transducer $\pi : \Sigma^* \to (2^{\Sigma^*})^*$. It takes as input the original execution trace and emits for each input event a set of all possible event sequences that can override that event. For example, a proxy that is allowed (but not obligated) to insert an event $b$ before an occurrence of event $a$ could be defined in such a way that for every $\sigma \in \Sigma^*$, $\pi(\sigma \cdot a) = \pi(\sigma) \cdot \{ba, a\}$. Note how the trace obtained from $\sigma$ is appended by a set of two possible event sequences: one where $b$ is inserted, and one where it is not. Since each "event" of the output is made of a set of sequences, we shall call them *sequence sets*. In this respect, it can be seen as a generalization of an earlier model, which was introduced to handle uncertainty and missing events as sets of possible worlds called "multi-events" [27].

It is important to stress that the proxy only models the enforcement capabilities of a monitor, irrespective of the actual property that is meant to be enforced. That is, if an enforcement monitor is allowed to remove any event from the trace, then the proxy will generate output traces where each event may or may not be present. Stated differently, the goal of the proxy is to generate all the possible modifications of the input trace that are potentially available to enforce a given property. An interesting feature of this model is to enable "non-standard" enforcement capabilities. For instance, classical delete automata can delete any event at any moment. Our abstract definition of a proxy could express a finer-grained capability, such as the fact that only successive $b$ events following an initial $b$ may be deleted (formalized as $\pi(\sigma \cdot bb) = \pi(\sigma) \cdot \{\epsilon, b\}$, and $\pi(\sigma) = \sigma$ otherwise). Since the proxy is not tied to a specific notation and has the leeway to output any sequence set it wishes, it offers a high capacity to precisely circumscribe available enforcement actions.

One can see in Figure 1 is that the output of transducer $\pi$ is not fed directly to the second phase of the enforcement pipeline. Rather, its output is post-processed so that traces of sequence sets are converted into a more compact representation called a *prefix tree*; each path in such a tree represents one possible sequence in the set. For the purpose of the enforcement pipeline, a special representation of these trees has been adopted, such that their contents can be transmitted in the form of a sequence of events. Let $\mathcal{V}\langle T\rangle$ denote the set of vectors of elements in $T$. For a given vector $v \in \mathcal{V}\langle T\rangle$, let $v[i]$ denote the element at position $i$ in that vector. Define $\mathcal{T} = \mathcal{V}\langle\mathcal{V}\langle\Sigma\rangle\rangle$ as the set of prefix tree elements, which are vectors of vectors of events. A prefix tree sequence is a trace $v_0, v_1, \ldots, v_n \in \mathcal{T}^*$, such that $v_0 = [[]]$, and for each $i \in [1, n]$:

$$|v_i| = \sum_{j=0}^{|v_{i-1}|} |v_{i-1}[j]|$$

The intuition behind this condition is that the $j$-th vector within a prefix tree element corresponds to the list of children attached to the $j$-th symbol in the prefix tree element that precedes it. As an example, the prefix tree in Figure 2 corresponds to the sequence of prefix tree elements [[]], [[$a$, $b$, $c$]], [[$a$, $b$], [$a$, $b$, $c$], [$a$]]. Note that a symbol may be $\epsilon$, so that a tree of given depth does not necessarily represent sequences of equal lengths. This representation makes it possible for a transducer to output a sequence of elements that represents the progressive construction of a prefix tree representing multiple event sequences. The task of box #5 in Figure 1 is precisely to receive each sequence set produced from $\pi$, and turn it into the appropriate prefix tree element.

## 3.2 Filtering of Valid Traces

The purpose of this setup becomes apparent in the next phase of the enforcement pipeline. The set of event traces generated by the proxy captures all the possible replacements of the original input trace. However, some of them are valid according to a given security policy, and others are not; one must therefore remove from the possible sequences produced by the proxy all those that violate the policy. To this end, this phase involves a monitor $\mu : \Sigma^* \to \mathbb{B}_4$, where $\mathbb{B}_4 \triangleq \{\top, \top^?, \bot^?, \bot\}$ is the set of 4-valued verdicts that follows the interpretation of RV-LTL [1]. In a nutshell, $\bot$ (resp. $\top$) is emitted for a prefix if the property is irremediably violated (resp. satisfied), while $\top^?$ indicates that the property is currently satisfied but could be violated in the future (and dually for $\bot^?$).

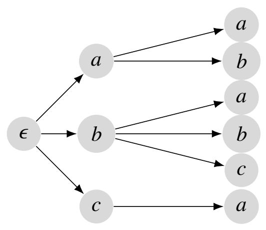


Fig. 2: Graphical representation of a prefix tree, for the multitrace $\{aa, ab, ba, bb, bc, ca\}$.

The task of filtering invalid traces is represented by box #6 in the pipeline. It receives as its input a sequence of prefix tree elements, and produces as its output a modified sequence of prefix tree elements, where any branches corresponding to prefixes violating the security policy are pruned out. If the monitor produces $\bot$ anywhere along a path, the node producing this verdict and all its descendants in the prefix tree are replaced by a placeholder $\Diamond$, indicating that these nodes should not be considered. If a path ends with the monitor producing $\bot^?$, the last node of that path is replaced by $\Diamond$. For example, suppose that the security policy imposes that a trace never start with $b$. In the tree of Figure 2, the leftmost $b$ node must therefore be deleted. In this particular case, the output of the filtering step would be the sequence of prefix tree elements [[]], [[$a$, $\Diamond$, $c$]], [[$a$, $b$], [$\Diamond$, $\Diamond$, $\Diamond$], [$a$]]. In contrast, a property stating that $a$ must eventually be followed by $b$ would result in the sequence [[]], [[$a$, $b$, $c$]], [[$\Diamond$, $b$], [$\Diamond$, $b$, $c$], [$a$]]. As a result, all remaining paths in the prefix tree correspond to prefixes of the trace that result in the monitor producing either $\top$ or $\top^?$.

Conceptually, it suffices to run a fresh instance of $\mu$ on each path of the induced prefix tree, and to remove a node (as well as all its descendants) as soon as $\mu$ appends $\bot$ to its output. However, the process needs to be done incrementally, since the contents of the prefix tree are produced one element at a time. Algorithm 1 shows how this can be done. The algorithm receives a vector of monitor instances and a prefix tree element of same size. The $\mu_{\sigma_i}$ represent the state of monitor $\mu$ after processing the paths ending in each leaf of the prefix tree, and the $v_i$ are the children events to be appended to each of these leaves. For each $\mu_{\sigma_i}$ and $v_i$, the algorithm iterates over each event $x$ in $v_i$ and adds to an output vector $m$ the monitor instance $\mu_{\sigma_i \cdot x}$, which is the result of feeding $x$ to $\mu_{\sigma_i}$. If the resulting output trace contains $\bot$, this path violates the security policy and the event $x$ is replaced by $\Diamond$. Otherwise, the event is added to the output vector, and the process repeats. The end result is a new pair of vectors $m$ and $v$, where $v$ is the filtered prefix tree element obtained from $[v_0, \ldots, v_n]$, and $m$ is the vector of monitor states for each leaf of this element.

**Algorithm 1** Incremental filtering

```
procedure FILTER([$\mu_{\sigma_1}, \ldots, \mu_{\sigma_n}$], [$v_0, \ldots, v_n$])
  $v \leftarrow [\,], m \leftarrow [\,]$
  for $i \leftarrow 1, n$ do
    $v' \leftarrow [\,]$
    for $x \in v_i$ do
      ADD($m, \mu_{\sigma_i \cdot x}$)
      if $\mu_{\sigma_i}(x)$ contains $\bot$ then
        ADD($v', \Diamond$)
      else if $i = n$ and $\mu_{\sigma_i}(x)$ ends with $\bot^?$
        ADD($v', \Diamond$) else ADD($v, v'$)
  return $(m, v)$
end procedure
```

**Algorithm 2** Output trace selection

```
1: procedure UPDATE([($\rho_{\sigma_1}, s_1$), ..., ($\rho_{\sigma_n}, s_n$)], [$v_0, \ldots, v_n$])
2:   $m \leftarrow [\,]$
3:   for $i \leftarrow 1, n$ do
4:     for $x \in v_i$ do
5:       $s = -\infty$
6:       if $x \neq \Diamond$ then
7:         $s \leftarrow$ LAST($\rho_{\sigma_1}(x)$)
8:       ADD($m, \rho_{\sigma_i \cdot x}$)
9:   return $m$
10: end procedure
```

As with the previous step, note that this operation is independent of the formal notation used to represent the security policy. It is applicable as long as the monitor is a computational entity outputting a sequence of elements in $\mathbb{B}_4$, and that stateful copies of itself can be cheaply produced.

## 3.3 Selection of the Optimal Output Trace

This final phase of the enforcement pipeline relies upon a special transducer, called the selector, which receives as input a sequence of prefix tree elements, and attempts to select the "optimal" one, based on a transparency condition. This phase involves a ranking transducer $\rho : \Sigma^* \rightarrow \mathbb{R}$, which assigns a numerical score to a trace. The principle of the selector is simple: each path in the filtered prefix tree is evaluated by $\rho$, and the path that maximizes the score is selected and returned as the output.

The operation of the selector, depicted in Figure 1 as box #7, is described by procedure UPDATE in Algorithm 2. This time, the procedure receives a prefix tree element $[v_0, \ldots, v_n]$ and a vector of pairs, each containing a ranking transducer instance $\rho_{\sigma_i}$ and the score $s_i$ this transducer has produced after processing $\sigma_i$. The algorithm then proceeds in a similar way as for FILTER: each transducer instance is fed with each child in sequence, and the updated instance and its associated score are added to the new vector $m$. Applying this procedure successively on each prefix tree element, and feeding the output vector $m$ back into the next call to UPDATE produces a vector, from which the output trace $\sigma_i$ can be chosen based on the highest score $s_i$ in all pairs.

## 3.4 Merging Valid vs. Corrected Trace

The last step of the pipeline, called the *gate* and represented by box #3, takes care of letting the input trace through as long as it does not violate the security policy, and to switch to the output of the enforcement pipeline only in case of a violation. This is why the gate receives as its inputs the original event trace, the output from the enforcement pipeline, as well as the verdict of the monitor $\mu$ for events of the input trace (box #2) that allows it to switch between the two. More precisely, the gate returns an input event directly if and only if $\mu$ does not produce the verdict $\perp$ or $\perp^?$ upon receiving this event. Otherwise, this event is kept into an internal buffer, and the gate awaits for an event or a sequence of events to be returned by the enforcement pipeline of box #4, which is output instead. As long as $\mu$ returns a false or possibly-false verdict, input events are added to the buffer and also fed to the enforcement pipeline. In such a way, the enforcement pipeline is allowed to ingest multiple input events and replace them by another sequence.

This mode of operation ends at the earliest occurrence of two possible situations. The first is if the monitor resumes returning either $\top$ or $\top^?$. In such a case, the input events in the buffer are deemed to be a safe extension of the ongoing trace, and are sent to the output. The second situation is if the enforcement pipeline produces a corrective sequence as its output. This indicates that the sequence of buffered input events must be discarded, and replaced by the output of the enforcement pipeline. After either of these two situations occur, the input buffer is cleared, and control is returned to the input trace.

However, doing so requires a form of feedback from the downstream gate to the upstream transducers, so that their internal state be consistent with the trace that has actually been output, and not the input trace that has been observed. To illustrate this notion, consider a simple security property stating that every $a$ event must be followed by a $b$. If the input trace is $ac$, the first $a$ event is output directly, as this prefix does not violate the policy. The next event, $c$, makes the prefix violate the policy; the gate therefore switches to the output of the enforcement pipeline. Suppose that this pipeline produces as its output the corrective sequence $bc$, which inserts a $b$ before the $c$. This sequence restores compliance with the policy, and events from the input trace can again be let through. However, the monitor $\mu$ of box #2, in charge of evaluating compliance of the trace, is still in an error state (having read $ac$); its verdict will therefore be incorrect for the subsequent incoming events.

This entails that one must be able to "rewind" $\mu$ and put it in the state it should after reading the real output trace ($abc$), so that it produces the correct verdict for the next events. It is the purpose of the feedback mechanism illustrated by the red arrows in Figure 1, and which we call *checkpointing*. Along with the transducer $\mu$ of box #2, a copy $\mu_\sigma$ is kept of that transducer, in the state it was after reading $\sigma$ (the "checkpoint"). Intuitively, $\sigma$ represents the sequence of events that have actually been output by the pipeline. As events are received, $\mu$ updates its internal state accordingly, but $\mu_\sigma$ is preserved. This copy is updated only when the downstream gate instructs it to, by providing a segment of newly output events $\sigma'$. When this occurs, both the checkpoint $\mu_\sigma$ and the internal state of $\mu$ are replaced by $\mu_{\sigma \cdot \sigma'}$. A similar feedback process occurs for the enforcement pipeline of box #4.

On its side, the gate notifies these transducers of a new checkpoint every time it outputs an event from the original input trace, or when a corrected segment from the enforcement pipeline is chosen instead. This makes sure that the whole system is always in sync with the contents of the actual output sequence.

### 3.5 Event Buffering

A final aspect of the architecture that needs to be discussed is the notion of buffering. The default behavior of the selector (box #7) is to keep accumulating prefix tree elements without producing an output, until a signal to pick a trace is given to it. This makes it possible to consider corrective actions generated by the proxy that may involve replacing a sequence of input events by another sequence of output events. However, the question remains as to how and when this signal should be emitted. The proposed architecture deliberately leaves this parameter open, enabling a user to select among various possibilities. We enumerate a few of them in the following.

The first is a greedy choice: every time the selector receives a prefix tree element, it picks the event that maximizes the evaluation of the ranking transducer (evaluated from the beginning of the trace) and immediately outputs it. The second strategy is to pick an output trace once a given threshold length is observed. Prefix tree elements are buffered until $k$ are received, after which the best path in the tree is selected (note that this path itself may be shorter than $k$ due to the presence of $\epsilon$ symbols). Yet another possibility is to buffer events until one of the traces reaches a threshold score. Finally, one last possibility is to base the decision to pick a trace on a condition evaluated on the prefix tree itself –for example by evaluating an auxiliary monitor $\delta : \Sigma^* \to \mathbb{B}_4$ on each path. As an example, one could decide to pick a trace whenever a specific event is observed in one of the paths.

## 4 Discussion

In this section, we discuss the advantages of the proposed enforcement model, illustrate its use with a simple use case, and describe a software implementation of these concepts.

### 4.1 Use Case

As an example, we consider a variant of the running example from Colombo *et al.* [8], which stems from a study of the remedial actions that can be taken to recover from violations of the terms of smart contracts. The example dictates the interaction between 3 types of principals: the casino, players and dealers. The casino provides a venue where dealers can set up games in which players can participate. Players then join by depositing a participation fee in the bank's account and guessing the result of a coin toss. After a prespecified time has elapsed, the dealer reveals the result and pays out the winners. A player who correctly guessed the parity of the number gets back twice his participation fee, paid by the dealer. If a player looses, he forfeits his participation fee, which is divided equally between the dealer and the casino.

The following set of events can occur in a trace of the casino: $NewGame(A)$ indicates the onset of a game by dealer $A$, $Bet(A)$ indicates that player $A$ has placed a bet. The occurrence of the $EndGame()$ event indicates the end of game, and enjoins the selector to cease buffering events, and take corrective action if needed. A payment from $A$ to $B$ will be noted by the event $Pay(A, B)$. All bets are worth are two dollars, and the $Pay()$ event transfers a single dollar. We write $Bet(\cdot)$ as a shorthand for $\bigvee_x Bet(x)$, for all players $x$ in the game. We likewise write $Pay(A, \cdot)$ (resp. $Pay(\cdot, A)$) for any payment in which principal $A$ is the recipient (resp. donor).

**Monitor** The policy that underpins this scenario is as follows: while a game is in progress, the balance of the dealer's account can never fall below the sum of the expected payouts. There are multiple ways this policy can be stated, but a particularly appropriate notation is through a system of stream equations over typed stream variables as defined in Lola [9]. A stream expression may involve the value of a previously defined stream. The language provides the expression $\text{ite}(b; s_1; s_2)$, which represents an if-then-else construct: the value returned depends on whether the predicate of the first operand evaluates to true. It also allows a stream to be defined by referring to the value of an event in another stream $k$ positions behind, using the construct $s[-k, x]$. If $-k$ corresponds to an offset beyond the start of the trace, value $x$ is used instead.

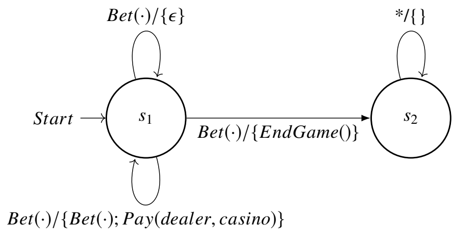
Defining the security policy using Lola becomes straightforward. The original event stream of casino events is first pre-processed to produce the Boolean streams $e$, $b$, $p^+$ and $p^-$, indicating whether an event is respectively an $EndGame$, a bet placed by a player, a payment from the player to the casino, or the reverse situation.

$$\begin{aligned}
t_1 &:= \text{ite}(e; 0; \text{ite}(b; t_1[-1, 0] + 2; t_1[-1, 0])) \\
t_2 &:= \text{ite}(p^+; t_2[-1, k] + 1; \text{ite}(p^-; t_2[-1, k] - 1; t_2[-1, k])) \\
\varphi &:= \text{ite}(\varphi[-1, \top], (t_2 - t_1) \geq 0, \bot)
\end{aligned}$$

The first equation defines a stream that keeps the count of the potential payouts to players. This counter is reset to 0 whenever a game ends; otherwise, it is incremented by 2 whenever a player places a bet, and keeps its value otherwise. The second equation keeps track of the dealer's balance, assuming the trace starts with an initial balance $k$. Stating that the potential payouts should never exceed the current balance then becomes the Boolean stream defined as $\varphi$, whose output can be used as the monitor verdict for the security policy.

**Proxy** The policy can be enforced by refusing (suppressing) bets when the dealer's assets are insufficient to cover them, or by lending (inserting) funds to the dealer's account. If a dealer is running multiple games simultaneously, the casino may also enforce the policy by prematurely ending some games, in the hopes that the winnings incurred by the dealer may allow him to accept further bets on other games. Refusing the bets submitted by a player incurs its own trade-off, since a player whose bets are consistently rejected may eventually take his business to a competing casino. For example, $\pi$ can be defined as a Mealy machine such as the one shown in Figure 3; depending on the current state and current input event, the machine may delete or insert other events.

**Selector** This policy exposes itself to several interrelated courses of actions, with the choices made by the monitor restricting its future course of action: canceling a game may turn off future patrons, refusing a bet incurs the loss of future revenue, reducing the monitor's freedom to reimburse players when the dealer defaults might further irritate some players. The enforcement pipeline will be forced to choose between these courses of action in order to attain one of several goals. This time, we opt for an extension of LTL called TK-LTL. We briefly recall the semantics of its important operators; the reader is referred to [21] for complete details.



$Bet(\cdot)/\{\epsilon\}$
$*/\{\}$
$Start$
$s_1$
$s_2$
$Bet(\cdot)/\{EndGame()\}$
$Bet(\cdot)/\{Bet(\cdot); Pay(dealer, casino)\}$

Fig. 3: Representation of a possible proxy enforcing the casino use case.

TK-LTL extends the semantics of LTL with several syntactic structures aimed providing a quantitative evaluation of different aspect of the trace. The feature upon which we rely the most is the counter $\widehat{C}^{v}_{\varphi}$, where $\varphi$ is an LTL formula and $v$ ranges over the truth values of LTL, returns the number of suffixes of the input trace for which the evaluation of $\varphi$ evaluates to $v$. Arithmetic operators or functions can be freely applied to the outputs of multiple counters over the same sequence to compute information about the trace. In addition to counters, the semantics of TK-LTL includes quantifiers. One of them is the propositional quantifier, and is written as $\mathcal{P}$. The formula $\mathcal{P}_{\sim k}\widehat{C}$ thus evaluates to $\top$ if the comparison $n \sim k$ holds where $n$ is the value returned by $\widehat{C}$. For example, let $\sigma = aaaba$ be a trace; the formula $\mathcal{P}_{=3}\widehat{C}^{\top}_{a}$ evaluates to $\top$ at positions $i = 3$ and $i = 4$, and to $\perp$ elsewhere.

The process of expressing the enforcement preorder is straightforward, and most of the possible requirements can be formulated as relatively simple formulas. For instance, the TK-LTL subformula $\widehat{C}^{\top}_{Bet(\cdot)}$ counts the total number of bets that are placed, and can be used as a transparency constraint if the casino's main concern is to maximize the total number of bets that are placed. A monitor that seeks to achieve this goal will thus avoid suppressing bet events from the input stream. Conversely, the formula $\widehat{C}^{\top}_{Pay(casino,\cdot)} - \widehat{C}^{\top}_{Pay(\cdot,casino)}$ expresses an alternative transparency requirement, namely maximizing gains for the casino.

## 4.2 Design Considerations

The modular design of the enforcement pipeline offers several advantages. Notably, it simplifies the creation of the monitor, since the process of manipulating the sequence is now separate from the process of the selecting a valid replacement. A main benefit of the method we propose is that the behavior of the enforcement monitor need not be coded explicitly. Instead, the behavior of the enforcement monitor is simply the result of the selector seeking to optimize the evaluation of the enforcement preorder.

The model also makes it possible to select the optimal replacement sequence, according to a criterion separate from the security policy, and which can be stated in a distinct formalism. The model also allows users to compare multiple alternative corrective enforcement actions, and select the optimal one with respect to an objective gradation. Finally, since the alteration of the input trace is done independently of its downstream verification for compliance with the policy, the model also does away with the need for a proof of correctness of the synthesized enforcement monitor, as is usually done in related works on the subject.

As we also stressed in Section 3, the proposed architecture is independent of the formal representation of each component. As a matter of fact, we deliberately chose three different notations for the proxy, monitor and ranking transducer of the casino use case to illustrate this feature. This flexibility makes it possible to support other types of enforcement requirements. For instance, consider a monitor whose objective is to produce a valid output that is as close to the input as possible. This is a fairly intuitive requirement, but difficult to implement using existing solutions. In the proposed framework, this requirement can be enforced by assigning a cost to each transformation performed by the monitor (adding an event or suppressing an event) and having the monitor minimize the overall enforcement cost for the entire sequence. Even more flexibility can be achieved by assigning different cost to each action as needed, or by assigning a different cost to suppression and insertion.

### 4.3 Implementation and Experimental Results

In the previous sections, we endeavored to describe the runtime enforcement model in an abstract way that is not tied to any specific system or formalism, and to give users the freedom of choosing the formal notation of their choice for each component of the pipeline. Nevertheless, a software implementation of this model has been developed as a Java library that extends the BeepBeep event stream processing engine [18].

This extension, which amounts to a little more than 2,600 lines of Java code, provides a new `Processor` class (the generic entity performing stream processing in the BeepBeep) called `Gate`. This class must be instantiated by defining four parameters. The first three are the transducers $\mu$, $\pi$ and $\rho$ representing the monitor, proxy and ranking transducer described earlier. In line with the formal presentation of Section 3, the pipeline makes no assumption about the representation of these three transducers. Any chain of BeepBeep processors is accepted, provided they have the correct input/output types for their purpose. For instance, an existing BeepBeep extension called Polyglot [16] makes it possible to specify the monitor using finite-state machines, LTL, Lola, or Quantified Event Automata [24], while another one can be used to define the ranking transducer by means of a TK-LTL expression. However, the user is free to pick from all of the available BeepBeep processors to form a custom chain for any of these components. Since every Processor instance in BeepBeep can create a stateful copy of itself at any moment, the checkpointing feature required by our proposed model is straightforward to implement.

The last parameter that must be defined is the strategy that decides how the filter and selector buffer and release events, as discussed in Section 3.5. Concretely, this is done by specifying a method named `decide`, which is called every time a new prefix tree element is received by the selector. By default, the enforcement monitor accepts an integer $k$ and picks an output trace after $k$ calls (with $k = 1$ corresponding to the immediate greedy choice); overriding this method produces a different behavior implementing another strategy. In the experiments, it was arbitrarily set to $k = 8$.

The rest of the operations are automated. Once a `Gate` is instantiated, it works as a self-contained processor which, internally, operates the pipeline described in Figure 1. To the end user, this processor can be used as a box receiving a sequence of events in $\Sigma$ and producing another sequence of events in $\Sigma$, which automatically issues corrected sequences when a policy violation occurs. It can be freely connected to other processor instances to form potentially complex computation chains.

To test the implemented approach, we performed several experiments made of a number of scenarios, where each scenario corresponds to a source of events, a property to monitor, a proxy applying specific corrective actions, a filter, and a ranking selector applying specific enforcement preorder. The set of experiments has been encapsulated into a LabPal testing bundle [17], which is a self-contained executable package containing all the code required to rerun them [26]. In addition to the *Casino* use case described earlier, our experiments include the following.

*Simple*: An abstract scenario where the source of events is a randomly generated sequence of atomic propositions from the alphabet $\Sigma = \{a, b, c\}$. Different proxies are considered for the purpose of the experiments: adding any event at any time, deleting any event at any time, adding/deleting only event $a$, or adding two events at a time. These proxies are meant to illustrate the flexibility of our framework to define possible corrective actions. Similarly, various policies are also considered: one corresponding to the LTL formula $\mathbf{G}\,(a \rightarrow (\neg b\,\mathbf{U}\,c))$, another that stipulates that events $a$ must come in pairs, and a last corresponding to the regular expression $(abc)^*$. Finally, the enforcement preorder in this scenario assigns a penalty (negative score) by counting the number of inserted and deleted events in a candidate trace. This leads the pipeline to favor solutions that make the fewest possible modifications to the input trace.

*File Lifecycle*: The second scenario is related to the operations that can be made on a resource such as a file, and is a staple of runtime verification literature [7]. A trace of events is made of interleaved operations open, close, read and write on multiple files. The policy is notable in that it is *parametric*: it splits the trace into multiple sub-traces (one for each file), and stipulates that each file follows a presrcibed lifecycle (read and write are allowed only between open and close, and no write can occur after a read). The scenario reuses a proxy and ranking transducer from *Simple*.

*Museum*: This example is taken from Drabik *et al.* [11], and illustrates quantified security policy enforcement. Events of the trace represent adults, children and guards going in and out of a museum. The policy specifies that access is forbidden for any children if no guard is currently present in the museum. The interest of this scenario lies in the possible variations for the proxy and enforcement preorder. A proxy can either insert a guard, prevent a guard from going out, or prevent a child from getting in. The possible enforcement preorders can be to minimize the number of modifications to the trace (as before), to maximize the number of children that enter the museum, or to minimize the number of time steps where guards are “idle” (present while no children are there).

For each variation of a scenario, we ran the enforcement pipeline on a randomly generated trace of length 1,000 of the corresponding type. The experiments are meant to assess the overhead, both in terms of running time and memory consumption, incurred by the presence of the proxy and the selector. A downloadable instance containing all the experiments described in this paper can be obtained online[1]. All the experiments were run on a Intel CORE i5-7200U 2.5 GHz running Ubuntu 18.04, inside a Java 8 virtual machine with the default 1964 MB of memory.

The results are summarized in Table 1. As one can see, the number of input events processed per second ranges in the hundreds to the thousands. Overall, one can conclude that the overhead incurred by the use of the pipeline is reasonable. For instance, in a real-world setting such as a blockchain, the limiting factor is more likely to be the number of transactions per second supported by the infrastructure itself; as a single example, the Ethereum network handles at most a few dozen transactions per second on the main net [4]. On its side, memory overhead remains relatively low with a few kilobytes, with a maximum demand of about 120 kB for a single scenario. Upon examination of the data, we observed that this corresponds to a single peak during the whole execution, with memory consumption otherwise remaining mostly below 10 kB.

Global overhead varies based on the actual combination of policy, proxy and ranking transducer. For instance, the $(abc)^*$ policy, when used on a proxy that only has the power to insert events into the trace, results in the slowest throughput. This scenario represents an extreme case since at any moment in the trace, a single next event is valid. Since the input trace is randomly generated, the probability that an input event not be the expected one is about $2/3$, meaning that the pipeline must perform a corrective action on almost every event.

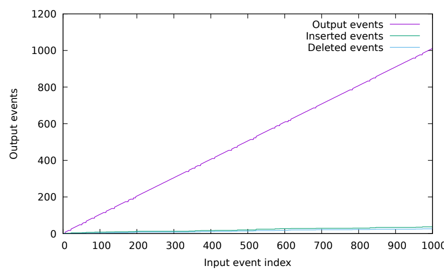
| **Event source** | **Policy** | **Proxy** | **Scoring formula** | **Throughput** | **Max memory** |
|---|---|---|---|---|---|
| Casino | Casino policy | Casino proxy | Maximize bets | 2380 | 9824 |
| | | | Maximize gains | 490 | 7976 |
| | | | Minimize changes | 2325 | 8814 |
| Files | All files lifecycle | Delete any | Minimize changes | 78 | 9580 |
| Museum | Museum policy | Museum proxy | Maximize children | 4347 | 9580 |
| | | | Minimize changes | 480 | 7984 |
| | | | Minimize idle guards | 1694 | 9580 |
| a-b-c | (abc)* | Delete any | Minimize changes | 628 | 9580 |
| | | Insert any | Minimize changes | 18 | 8692 |
| | After a, no c until b | Delete any | Minimize changes | 869 | 8236 |
| | | Insert any | Minimize changes | 67 | 119076 |
| | | Insert any b | Minimize changes | 485 | 10344 |
| | Stuttering a's | Delete any | Minimize changes | 952 | 9580 |
| | | Insert any | Minimize changes | 602 | 9396 |

Table 1: Summary of throughput (in events/sec.) and maximum memory consumption (in bytes) for each scenario.

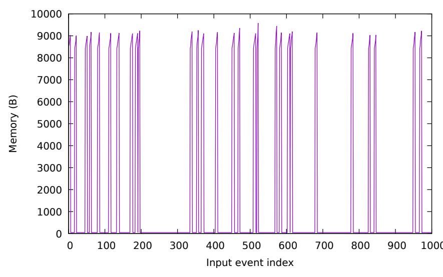
The action of a proxy can also be examined in further detail. Figure 4a shows the cumulative number of deleted, inserted and output events produced as the input trace is being read, for a variant of the museum scenario. Although difficult to see due to the scale of the plot, the output event line increases in an irregular staircase pattern. This is caused

[1] `https://github.com/liflab/multitrace-enforcement-lab`

by the fact that the gate withholds events at moments where the policy is temporarily violated. One can also observe that, for this scenario, the enforcement pipeline inserts and deletes events in a relatively equal (and small) proportion.



(a) Input vs. output events



(b) Memory consumption

Fig. 4: Runtime statistics for the execution of an enforcement pipeline on a variation of the museum scenario.

On its side, Figure 4b shows the memory used by the pipeline at each point in the execution. Memory remains near zero as long as the input trace does not violate the property; as a matter of fact, these flat regions exactly match the locations in Figure 4a where no change occurs on both inserted and deleted events. The memory plot also shows spikes, which correspond to the moments in the trace where the enforcement pipeline kicks in and starts generating possible corrected sequences. Once one such sequence is chosen and emitted, all data structures are cleared and memory usage drops back to zero. These observations are consistent with the expected operation of the pipeline described in Section 3.

## 5 Conclusions

In this paper, we presented a flexible runtime enforcement framework to provide a valid replacement to any misbehaving system and guarantee that the new sequence is the optimal one with respect to an objective criterion we call transparency constraints. A proxy interposed between the input sequence and the monitor is used to generate all the possible replacements. A monitor then eliminates invalid options, while a selector identifies the optimal replacement sequence with respect to a transparency constraint, separate from the security policy. We described a novel formalism to state this constraint; the implementation of these concepts as an extension leveraging the BeepBeep event stream processing engine, and run through a range of different scenarios, has shown that the enforcement of a property can be done dynamically at runtime without the need to manually define an enforcement monitor specific to the use case considered.

Therefore, the precise behavior of the pipeline can be seen as being emergent from the interplay of its components. Moreover, we stressed how this modular design makes it possible to easily replace any element of the framework (policy, proxy, preorder) by another. As a matter of fact, each individual transducer used in the scenarios benchmarked in Section 4.3 requires at most a few dozen lines of code. This genericity opens the way to the future study of a broad range of enforcement mechanisms under a uniform formal framework, and to a more detailed comparison of their respective advantages. It should also be mentioned that, for many of the scenarios we experimentally tested, most of the proxies that are considered are given very large license to modify the trace, for example by inserting or deleting any event at any moment. This obviously has an impact on runtime overhead, as it causes the generation of a large number of potential corrected traces. One could consider proxies with tighter enforcement capabilities.

In addition, our model can be subject to multiple extensions and enhancements. For instance, it can be extended to evaluate more than one transparency requirement over the traces. The pipeline of Figure 1 can be modified by considering multiple ranking transducers, where each transducer evaluates a specific transparency requirement and assigns a numerical score to each output trace of the proxy based on the enforcement preorder. One could also consider relaxing the classical definition of transparency, and allow modifications to a trace that are not triggered only by hard violations of a policy.

Finally, the treatment of partial and ambiguous events known as gaps that may be present in an input trace could be an area of future research. A proxy could be used to model different types of data degradation in order to fill in the gaps in the trace with all potential events as we did in [27]. The same or another proxy could be used to enforce the desired policy, then the filter filters the traces and the selector quantifies the traces and chooses the optimal one. As a result, our framework will be useful in a variety of situations including enforcing policies over corrupted logs or any other data source with insufficient information.

## References

1. Bauer, A., Leucker, M., Schallhart, C.: Comparing LTL semantics for runtime verification. J. Log. Comput. 20(3), 651–674 (2010)
2. Bauer, L., Ligatti, J., Walker, D.: More enforceable security policies. In: In Foundations of Computer Security (2002)
3. Beauquier, D., Cohen, J., Lanotte, R.: Security policies enforcement using finite and pushdown edit automata. Int. J. Inf. Sec. 12(4), 319–336 (2013)
4. Betti, Q., Montreuil, B., Khoury, R., Hallé, S.: Smart Contracts-Enabled Simulation for Hyperconnected Logistics, pp. 109–149. Springer International Publishing, Cham (2020)
5. Bielova, N., Massacci, F.: Do you really mean what you actually enforced? - edited automata revisited. Int. J. Inf. Sec. 10(4), 239–254 (2011)
6. Chang, E., Manna, Z., Pnueli, A.: The safety-progress classification. In: Bauer, F.L., Brauer, W., Schwichtenberg, H. (eds.) Logic and Algebra of Specification. pp. 143–202. Springer Berlin Heidelberg, Berlin, Heidelberg (1993)
7. Chen, F., Meredith, P.O., Jin, D., Rosu, G.: Efficient formalism-independent monitoring of parametric properties. In: ASE. pp. 383–394. IEEE Computer Society (2009)
8. Colombo, C., Ellul, J., Pace, G.J.: Contracts over smart contracts: Recovering from violations dynamically. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11247, pp. 300–315. Springer (2018)

9. D'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: LOLA: runtime monitoring of synchronous systems. In: TIME. pp. 166–174. IEEE Computer Society (2005)
10. Dolzhenko, E., Ligatti, J., Reddy, S.: Modeling runtime enforcement with mandatory results automata. Int. J. Inf. Secur. 14(1), 47–60 (Feb 2015)
11. Drábik, P., Martinelli, F., Morisset, C.: Cost-aware runtime enforcement of security policies. In: Jøsang, A., Samarati, P., Petrocchi, M. (eds.) STM. LNCS, vol. 7783, pp. 1–16. Springer (2012)
12. Drábik, P., Martinelli, F., Morisset, C.: A quantitative approach for inexact enforcement of security policies. In: Gollmann, D., Freiling, F.C. (eds.) ISC 2012. LNCS, vol. 7483, pp. 306–321. Springer (2012)
13. Falcone, Y., Mariani, L., Rollet, A., Saha, S.: Runtime failure prevention and reaction. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification - Introductory and Advanced Topics, LNCS, vol. 10457, pp. 103–134. Springer (2018)
14. Falcone, Y., Mounier, L., Fernandez, J.C., Richier, J.L.: Runtime enforcement monitors: Composition, synthesis, and enforcement abilities. Form. Methods Syst. Des. 38(3), 223–262 (Jun 2011)
15. Fong, P.W.L.: Access control by tracking shallow execution history. In: S&P 2004. pp. 43–55. IEEE Computer Society (2004)
16. Hallé, S., Khoury, R.: Writing domain-specific languages for BeepBeep. In: Colombo, C., Leucker, M. (eds.) RV. LNCS, vol. 11237, pp. 447–457. Springer (2018)
17. Hallé, S., Khoury, R., Awesso, M.: Streamlining the inclusion of computer experiments in a research paper. Computer 51(11), 78–89 (2018)
18. Hallé, S.: Event Stream Processing With BeepBeep 3: Log Crunching and Analysis Made Easy. Presses de l'Université du Québec (2018)
19. Hamlen, K.W., Morrisett, J.G., Schneider, F.B.: Computability classes for enforcement mechanisms. ACM Trans. Program. Lang. Syst. 28(1), 175–205 (2006)
20. Khoury, R., Hallé, S.: Runtime enforcement with partial control. In: García-Alfaro, J., Kranakis, E., Bonfante, G. (eds.) FPS 2015. LNCS, vol. 9482, pp. 102–116. Springer (2015)
21. Khoury, R., Hallé, S.: Tally keeping-LTL: An LTL semantics for quantitative evaluation of LTL specifications. In: IRI 2018. pp. 495–502. IEEE (2018)
22. Khoury, R., Tawbi, N.: Corrective enforcement: A new paradigm of security policy enforcement by monitors. ACM Trans. Inf. Syst. Secur. 15(2), 10 (2012)
23. Mallios, Y., Bauer, L., Kaynar, D., Ligatti, J.: Enforcing more with less: Formalizing target-aware run-time monitors. In: Proceedings of the International Workshop on Security and Trust Management. pp. 17–32 (Sep 2012)
24. Reger, G., Cruz, H.C., Rydeheard, D.E.: MarQ: Monitoring at runtime with QEA. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 596–610. Springer (2015)
25. Schneider, F.B.: Enforceable security policies. ACM Trans. Inf. Syst. Secur. 3(1), 30–50 (Feb 2000)
26. Taleb, R., Hallé, S., Khoury, R.: Benchmark measuring the overhead of runtime enforcement using multi-traces (LabPal package) (2022), DOI: `10.5281/zenodo.5976001`
27. Taleb, R., Khoury, R., Hallé, S.: Runtime verification under access restrictions. In: Bliudze, S., Gnesi, S., Plat, N., Semini, L. (eds.) FormaliSE@ICSE 2021. pp. 31–41. IEEE (2021)
28. Talhi, C., Tawbi, N., Debbabi, M.: Execution monitoring enforcement for limited-memory systems. In: PST. PST '06, Association for Computing Machinery, New York, NY, USA (2006)