A Modular Pipeline for Optimal Enforcement of Security Properties at Runtime

Rania Taleb[1*], Sylvain Hallé[1] and Raphaël Khoury[1]

[1*]Laboratoire d'informatique formelle Université du Québec à Chicoutimi, Canada .

*Corresponding author(s). E-mail(s): rania.taleb1@uqac.ca;
Contributing authors: shalle@acm.org; rkhoury@uqac.ca;

Abstract

Runtime enforcement ensure the respect of a user-specified security policy by a program by providing a valid replacement to any misbehaving sequence of events that may occur during that program's execution. 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. Furthermore, the current design of runtime monitors imposes a substantial burden on the designer, since the entirety of the monitoring task is accomplished by a monolithic construct, usually an automata-based model. In this paper, we propose a new modular model of enforcement monitors, in which the tasks of altering the execution, ensuring compliance with the security policy and selecting the optimal replacement are split in three separate modules, which simplifies the creation of runtime monitors. We implement this approach with 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 a security enforcement paradigm that prevents a monitored program from misbehaving by intervening as needed to enforce a user-specified security policy [33]. Unlike runtime *verification*, the monitor is intended to provide a proper alternative for any misbehaving trace rather than merely signalling a violation. The growing popularity of smart contracts has prompted fresh interest in runtime enforcement [22]. Smart contracts can't be changed after they've been deployed, therefore the only way to deal with unexpected behaviour is to enforce it at runtime.

Under this paradigm, the execution of a program is abstracted as a sequence of events, called actions, while the desired security policy is abstracted as set of valid sequences, called the property. Enforcement is commonly performed by way of an *enforcement monitor*: a formal model, such as a transducer, that receives as input the original sequence of the program and outputs an alternate execution sequence that provably respects the property. The enforcement monitor is generally tasked with ensuring conformity with two basic principles: *soundness* and *transparency* [42, 64]. Soundness imposes that the output of the monitor must respect the underlying security policy; while, 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 other words, the monitoring process cannot alter the semantics of valid

traces. Further research also suggested that the monitor should also be limited in the changes that it performs on invalid executions as well, otherwise, almost any property can be enforcement, but not necessarily in a manner that is useful or desirable [14, 49]. This observation bears a pivotal consequence on the conduct of the monitor: in cases where multiple possible paths to enforcing the property are possible, the monitor should be required to select the *optimal* one, w.r.t. some gradation of executions.

In this context, the monitor is in effect a mathematical structure tasked with performing the entirety of the enforcement process: reading the input, transforming it through a process of substitutions, insertions, deletions and/or truncations, ensuring compliance of the resulting output trace with respect to both soundness and transparency, and ensuring that the output is the optimal. The monitor must thus encapsulate the desired security policy, the desired gradation of solution, and limitations (i.e. memory or computational limitations) imposed on the monitor's ability to transform input sequences. This monolithic design incurs a number of disadvantages. In particular with respect to difficulties in generating a correct monitor for a given security policy. In addition, elaborate proofs are often required to ensure that the output of the monitor is indeed sound and transparent (see e.g. [49]).

The present paper offers an alternative solution to the problem: we introduce 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. This, in turn, makes it easier to reason about the behaviour of the whole pipeline. 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.

This paper is an extended version of a paper that was presented at the 14th International Symposium on Foundations & Practice of Security. Compared with the original version, the main changes are:

1. An expanded explanation of several related concepts, such as runtime verification and enforcement, and an expanded state of the art, which provides a complete primer to runtime enforcement and related concepts.
2. An additional section, (Section 3—- Formal Foundations), that details the mathematical underpinnings of traces, events, policies, truth domains, and other related concepts. This section notably details the various ways a security policy can be specified.
3. An additional section, (Section 4—- Altering Input Traces), which details the different types of proxies that can be created, and relates the notion of proxies to the existing literature in runtime enforcement.
4. An additional section, (Section 5—- Producing Corrected Traces), which provides a formal definition and a thorough discussion of the two keys concepts that underpin our notion of enforcement, namely correcting the input sequence and selecting the optimal corrective course of action.
5. : An additional use case and examples, which helps illustrates key insights about our approach.
6. A important addition is Section 7.4-7.6, in which empirically compares the effect of various enforcement strategies and scoring functions for the same policy and input sequence. We further compare the overhead of enforcing the property using our pipeline with that of using a conventional automaton model.
7. Numerous other sections were rewritten for clarity and expanded upon.

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 provide formal definitions

for the main components of a runtime enforcement framework: *policies* and *transducers* with some examples. Section 4 introduces the notion of proxy as a transducer and describes categories of proxies used in the literature. Section 5 describes the notion of trace correction as an interplay between a monitor output and the alterations made by proxy to correct a trace. Equipped with these notions, we present our model of runtime enforcement in section 6 while illustrating the flexibility of the approach with two use cases adapted from the literature. Section 7 presents a concrete implementation of our pipeline and different categories of proxies as extensions of the BeepBeep event stream processing library [40] and provides comparison and discussion of the obtained results. Concluding remarks are given in Section 8.

2 State of the Art in Runtime Enforcement

Runtime verification [4, 44, 55] is is the discipline of computer science where an object called *monitor* is used to observe the behaviour of another program, its target, in order to detect a potential violation of a user-defined security policy and emit a true verdict if the policy is satisfied and a false verdict if it is violated. It is related to the problem of runtime enforcement [31, 43, 59, 64], which additionally seeks to react to any observed violation in such a way as to correct recover from it, for example by modifying the execution or skipping execution steps. In both cases, the execution of the target system is abstracted as a sequence of program events, termed the input sequence. The security policy is usually, but not necessarily, a predicate over individual sequences, in which case, the terms policy and property can be used interchangeably.

These security mechanisms have been applied successfully to the field of telecom, notably to ensure conformity with cryptographic protocols [5, 70], other network protocols [2, 41] and to monitor client-server communications in online stores [65].

Of particular importance in this paper is an enforcement monitor, which is defined as a processing unit that can transform an input sequence of events into another sequence. An enforcement monitor is said to satisfy the *soundness* condition if its application on an input sequence always results in an output sequence that satisfies a given

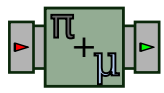


Figure 1: A symbolic representation of an enforcement monitor. Events enter on the left-hand side and a corrected version of the input stream is produced on the right-hand side.

policy. An enforcement monitor can be represented graphically as in Figure 1; symbols π and μ represent entities called the "proxy" and the "policy monitor", which we shall define in later sections. 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 [10, 49, 64]. 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). The design and capabilities of the proxy will naturally have a profound influence on the enforcement power of the complete pipeline [50]. A thorough survey of runtime enforcement, stressing its connection to runtime verification, is given by Falcone *et al.* [33].

2.1 Monitor Capabilities

In his initial formulation, Schneider [64] 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. Truncation is the only remedial path available to such monitors, and the output of the monitor is thus necessarily the longest prefix of the input sequence that satisfies the desired property. These security automaton are limited to enforcing safety properties [43], which states that nothing bad happens during program execution. However, the enforcement power of the monitor can be extended if it has access to the results of a static analysis of its target's code. Such an analysis allows the monitor to build a model of

the target program's possible behaviour, enhancing the mechanisms' enforcement power [18].

Ligatti *et al.* [10] 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. This categorization gives rise to a hierarchy of proxies with those that have edition capabilities being the most powerful. Suppression and insertion monitors have capabilities that are orthogonal to one another, and truncation monitors are the least powerful one. Another characterization, in which some events lie beyond the control of the monitor, was proposed by Khoury *et al.* [46].

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. This is a slightly more involved version of a property already proposed by Schneider [64]. An execution violating this policy could be one in which the file is read, and a *send* action subsequently occurs, without an intervening *log* action. 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).

Later research distinguishes further subcategories of these monitors, derived from finer limitations on the ability of the monitor to alter the input sequence. For instance, the monitor may be limited to inserting events that have previously been suppressed [13, 15] or limited in its capacity to insert or suppress certain events [25, 47]. After all, the insertion or suppression of some events may be beyond the control of the monitor for a variety of reason, such as computability constraints or because performing the underlying program actions requires encryption keys. The presence of uncontrollable actions brings a case-by-case subtlety to question of enforceability.

In this line of research, the monitor is usually modelled as a finite state machine, which dictates its behaviour 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.* [31] showed that a finer automaton model, with explicit store and dump operations, can enforce policies in the *response* class from the safety-progress classification [19]. 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.* [68] study the enforcement power of monitors with bounded memory; Fong *et al.* [34] study a monitor that only records the shallow history (i.e. the unordered set of events) of the execution, while Beauquier *et al.* [11] 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 [26].

2.2 Transparency Constraints

Several authors also considered how the license given to the monitor to alter valid and invalid sequences affects its enforcement power, and in fact they have found that this aspect of monitoring is in some way even more consequential than the monitor's capabilities when delineating the set of properties that are enforceable by a monitor. 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 [10], which can lead to undesirable behaviour. 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. Indeed, when using the definition of enforcement given above, a monitor is said to enforce a property as long as it can replace an invalid sequence with *any* valid sequence, even one completely unrelated to the input the monitor has received. For example, Bielova *et al.* create sub-classes that further constrain the monitor's handling of invalid executions [14]. 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. In the example given above, the transparency constraint could be based on the number of completed open-closed pairs. This would prevent the monitor from deleting valid parts of an otherwise invalid sequence.

Khoury *et al.* also consider constraints on invalid sequences, and introduce the notion of "gradation" of solutions [49]. 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 seekg 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.* [28] 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 $f : \Sigma^* \to \mathbb{R}$, 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 [31], 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 Formal Foundations

In this section, we provide formal definitions and examples for two main components of a runtime enforcement framework: the *security policy*, whose task is to define what is a "correct" input, and the *transducer*, which has the power of transforming an input into a modified output.

In the present context, inputs and outputs will be taken as sequences of arbitrary data objects called *events*. To this end, let Σ be a finite or countably infinite set of elements called *events*. For simplicity, we focus on atomic events, but the framework presented in this paper is easily generalized to parameters of data-bearing events. The set of all finite sequences from Σ, also called *traces*, is given as Σ^*. Given a trace $\overline{\sigma} \in \Sigma^*$, we use the notation $\overline{\sigma}[i]$ to range over the elements of $\overline{\sigma}$, where i represents the event at the i-th position (the first event is at $i = 0$). The notation $\overline{\sigma}[i..]$ denotes the remainder of the sequence starting from action $\overline{\sigma}[i]$ while $\overline{\sigma}[..i]$ denotes the prefix of $\overline{\sigma}$, up to its i-th position.

The concatenation of two sequences $\overline{\sigma}$ and $\overline{\sigma}'$ is given as $\overline{\sigma} \cdot \overline{\sigma}'$. The empty sequence is denoted ϵ, and $\overline{\sigma} \cdot \epsilon = \epsilon \cdot \overline{\sigma} = \overline{\sigma}$. As usual, the notation $\overline{\sigma}' \preceq \overline{\sigma}$ denotes that $\overline{\sigma}'$ is a prefix of $\overline{\sigma}$. Given a sequence $\overline{\sigma}$ and a set $S \subset \Sigma^*$, we override notation by letting $\overline{\sigma} \cdot S$ denote the set: $\bigcup_{\overline{\sigma}' \in S} \{\overline{\sigma} \cdot \overline{\sigma}'\}$. In the same way, if S and S' are two sets of traces, $S \cdot S'$ is defined as $\bigcup_{\overline{\sigma} \in S} \overline{\sigma} \cdot S'$.

3.1 Security Policies

Based on this elementary notation, we can now provide a formal definition of security policies. Although we use the term "security" to call these policies, they can be more broadly interpreted as the definition of what constitutes a valid input. This notion can vary depending on the context; for example, a policy could represent the expected ordering of operations defined by some network protocol, and valid inputs according to this policy would correspond to protocol-compliant sequences.

3.1.1 Definition

A security policy is a subset $\Phi \subseteq \Sigma^*$ of sequences called the *valid* sequences. For example, given an abstract alphabet $\Sigma = \{a, b\}$ made of only two events, the policy stated informally as "a must be the first event" corresponds to the set $\{a, aa, ab, aaa, aab, aba, abb, \dots\}$ made of all finite traces that start with a. Typically, a policy circumscribes an infinite subset of Σ^*, although this is not a requirement.

This set Φ induces a function $\varphi : \Sigma^* \to \mathbb{B}_4$, which associates to every trace a value called the *verdict*. The set $\mathbb{B}_4 = \{\top, \top^?, \bot_?, \bot\}$ corresponds to four possible "Boolean" outcomes, with $\top$ and $\bot$ respectively meaning "true' and "false". The remaining two values, which can intuitively be interpreted as "possibly true" and "possibly false", represent a form of uncertainty in the verdict. Formally, function φ is defined as follows:

$$\varphi(\overline{\sigma}) = \begin{cases} \top & \text{if } \overline{\sigma} \in \Phi \wedge \forall \overline{\sigma}' \in \Sigma^*, \overline{\sigma} \cdot \overline{\sigma}' \in \Phi \\ \top^? & \text{if } \overline{\sigma} \in \Phi \wedge \exists \overline{\sigma}' \in \Sigma^* \text{ s.t. } \overline{\sigma} \cdot \overline{\sigma}' \notin \Phi \\ \bot_? & \text{if } \overline{\sigma} \notin \Phi \wedge \exists \overline{\sigma}' \in \Sigma^* \text{ s.t. } \overline{\sigma} \cdot \overline{\sigma}' \in \Phi \\ \bot & \text{if } \overline{\sigma} \notin \Phi \wedge \forall \overline{\sigma}' \in \Sigma^*, \overline{\sigma} \cdot \overline{\sigma}' \notin \Phi \end{cases}$$

When $\varphi(\overline{\sigma})$ returns true ($\top$), it indicates that the policy is currently satisfied, and will remain so forever, regardless of events that can be appended to it. For example, a simple policy stating that "event a eventually occurs" becomes true for a trace as soon as it contains a. In the same way, when $\varphi(\overline{\sigma})$ returns false ($\bot$), it indicates that the policy is currently violated and is irremediably so. The policy stating that "event a should never occur" becomes false for a trace as soon as it contains a, and whatever events are appended at the end of $\overline{\sigma}$ cannot change this fact.

As one may guess, the definition of the remaining two possible verdicts suggests that the fate of the security policy depends on what may come after. Verdict "possibly true" ($\top^?$) indicates that $\overline{\sigma}$ currently satisfies the policy, but that there exists a continuation of that trace that does not belong to the security policy. For instance, the policy stating that "a must not occur" is satisfied by the trace consisting of the single event b, but there exists an extension of that trace which does not belong to Φ (namely the trace ba). Conversely, the policy stating that "a must eventually occur" is not satisfied by the trace consisting of the single event b, but there exists an extension of that trace which does belong to Φ (again, the trace ba). Consequently, this trace would be associated to the "possibly false" ($\bot_?$) verdict. One can then define the Boolean connectives on these four values, by assuming a total order on $\mathbb{B}_4$ such that $\bot < \bot_? < \top^? < \top$. Then for $x, y \in \mathbb{B}$, we have that $x \wedge y \triangleq \min(x, y)$, $x \vee y \triangleq \max(x, y)$. Negation is defined as usual for $\top$ and $\bot$, and further $\neg\bot_? = \top^?$.

A verdict that is either true or possibly true will be called a *positive* verdict; similarly, a verdict that is either false or possibly false will be called a *positive* verdict. A verdict that belongs to $\{\top, \bot\}$ is said to be *definitive*, otherwise it is called *uncertain*.

3.1.2 Notations for Security Policies

Our previous definitions are agnostic with regard to the precise way in which set Φ is defined. However, there exist classical ways of representing a security property.

Regular expressions are a popular declarative language for describing sets of strings in computer science [4]. The runtime verification community additionally relies upon temporal logics. Informally, a regular expression is a pattern that specifies a regular language and thus expresses a policy. As an example, consider the policy stating that a red light should be immediately followed by a green light. Using regular expression operators, this can be expressed as follows:

$$((\text{green} \mid \text{yellow})^* \ \text{red}^+ \ \text{green})^*$$

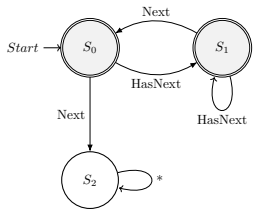


Figure 2: A graphical representation of a finite-state automaton representing the constraint that *Next* cannot be called before calling *HasNext* first [54].

Another method is to use concepts from formal language theory in which the patterns of symbols are described using different forms of **finite-state automaton**. We recall that a finite-state automaton is a quadruplet $M = \langle S, s_0, \delta, S_F \rangle$ where S is a set of states, $s_o \in S$ is the initial state, $\delta : S \times \Sigma \to S$ is the transition function, and $S_F \subseteq S$ is the set of final or accepting states. These automaton can be represented in the form of a graph, as is illustrated in Figure 2.

A finite-state automaton can give an immediate incremental checking procedure using their transition function. Starting from the initial state, for every atomic proposition, the machine moves from the current state to the next state. The verdict associated to the trace prefix can be deduced from the possible paths ahead of the current state s state in the automaton. If $s \in S_F$, the verdict is $\top$ if only states in S_F are reachable from it, and $\top^?$ otherwise. If $s \notin S_F$, the verdict is $\bot$ if only states in $S - S_F$ are reachable from it, and $\bot_?$ otherwise. Finite-state automaton admit several variants, such as multi-track automaton [71]. This mechanism is also widely used for runtime enforcement [10].

Finally, **Linear Temporal Logic** (LTL) [61] can also be used to express policies. A specification property φ is represented as an LTL formula using several operators such as the negation operator ($\neg$), unary temporal operators (e.g. **G**, **F**, **X**) where Gφ stands for *Globally* and means that a formula φ should *always* hold, Fφ stands for *Finally* and means that a formula φ should *finally* hold and Xφ stands for *Next* meaning that a formula φ should hold in the *next* state. We also have binary temporal logic operators such as U stands for *Until* where φ_1 U φ_2 means that φ_1 has to hold at least until φ_2 becomes true, which must hold at the current or a future position, the conjunction operator ($\wedge$) and the disjunction operator ($\vee$). An example of LTL formula can be: $\mathbf{G}\,\neg a \wedge \mathbf{F}\,b$ stating that a should never hold and b must finally hold.

Several extensions of LTL also exist [6, 8, 9, 32, 56] as well as TK-LTL [48] which is described in section 6.2.2. Note that a combination of regular expressions and temporal logic was also used in several approaches [3, 7, 45]. Other formal specification languages include the Temporal Stream-Based Specification Language (TeSSLa) [23], Metric Temporal Logic [53] and Lola [24] specification, which defines is a set of equations over typed stream variables and describes the computation of output streams from a given set of input streams. It is used in section 6.2.2.

3.2 Transducers

Given two sets of events Σ_1, and Σ_2, a trace transducer is a function $\tau : \Sigma_1^* \to \Sigma_2^*$, with the added condition that for every $\overline{\sigma}, \overline{\sigma}' \in \Sigma^*$, $\overline{\sigma}' \preceq \tau(\overline{\sigma})$ implies $\overline{\sigma}' \preceq \tau(\overline{\sigma} \cdot x)$ 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 $\overline{\sigma} \in \Sigma_1^*$, we define $\tau_{\overline{\sigma}} : \Sigma_1^* \to \Sigma_2^*$ as $\tau_{\overline{\sigma}}(\overline{\sigma}'') = \tau(\overline{\sigma} \cdot \overline{\sigma}'')$. Intuitively, $\tau_{\overline{\sigma}}$ is a device abstracting the "internal state" of the transducer τ after ingesting the events from the prefix $\overline{\sigma}$.

A transducer is said to be k-bounded if for every $\overline{\sigma} \in \Sigma^*$, $|\overline{\sigma}| \leq k \cdot |\tau(\overline{\sigma})|$. This means that for every new input event, the transducer adds to its output at most k events. The transducer is said to be k-monotonic if it produces exactly k output events for each input event.

One can lift a policy Φ and its associated verdict function ϕ into a transducer $\widehat{\phi} : \Sigma^* \to \mathbb{B}_4^*$, defined as follows, for all $i \in \mathbb{N}$:

$$\widehat{\varphi}(\overline{\sigma})[i] \triangleq \varphi(\overline{\sigma}[..i])$$

In other words, the i-th output event of $\widehat{\varphi}$ corresponds to the verdict associated to $\overline{\sigma}$ after reading its first i events. This transducer is called the *monitor*: it can be seen as an entity observing the input sequence of events, and updating after each event the verdict associated to that sequence according to the underlying policy. By virtue of the definition of φ, a monitor's output consists of a sequence of

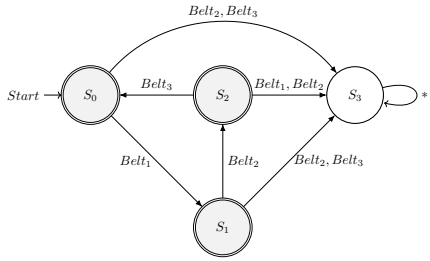


Figure 3: Parcel Dispatcher Property Automaton [30].

uncertain verdicts (positive, negative, or a mixture of both), and may eventually settle on a definitive true or false verdict, after which it never changes.

3.3 Examples

Next and HasNext pattern Policy: The standard Java API defines a number of interfaces in which the flow of methods invoked on objects must follow specific patterns in order for them to be utilized correctly [1, 16]. Such patterns are described in the documentation using a number of rules where the violation of these rules can cause the program to misbehave, or throw an exception. A common example of this situation concerns the methods *HasNext* and *Next* of the iterator interface. The proper use of an iterator stipulates that one should never call method next() before first calling method HasNext(). The correct ordering of these calls can be expressed by a finite state machine shown in Figure 2, and can also be described by an LTL formula as follows:

$$\mathbf{G}(\text{Next} \rightarrow \mathbf{X}\ \text{HasNext})$$

Fair Parcel Dispatcher Policy: In their work, Falcone *et al.* [30] considered a dispatcher who can accept parcels and distribute them to three conveyor belts. The initial behaviour of the dispatcher is arbitrary, in the sense that the dispatcher can move a parcel to any belt. The desired property states that *"the dispatcher is fair, in the sense that it distributes the parcels on the belts one after the other in a specified order"*. The specification is modelled using the automaton given in Figure 3, which defines the fair re-partition among three conveyor belts following the order $Belt_1$, then $Belt_2$, then $Belt_3$. The alphabet of the property is $\Sigma = \{Belt_1, Belt_2, Belt_3\}$. The accepting states are S_0, S_1 and S_2.

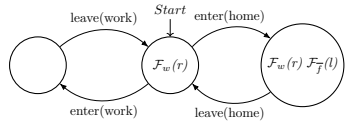


Figure 4: A Finite State Machine for the OSN policy [58].

Privacy Policies in Online Social Networks: On Online Social Networks (OSN) [58], there are many context and time dependent dynamic policies that can be expressed using static operators as well as represented using a deterministic automaton with transitions labelled by events which the online social network can perform. For instance, the policy *Co-workers cannot see my posts while I am not at work, and only family can see my location while I am at home* can be expressed using the static policy operator $\mathcal{F}_g(x)$ to denote that anyone in group g is forbidden from performing action x, where x can refer to the forbidden action such as posting, seeing a location, etc.), and $\mathcal{F}_{\overline{g}}(x)$ to denote the complement of a group of users g. Then the policy while not being at work can be expressed as $\mathcal{F}_{co-workers}(\textit{read-post})$, and the policy when not at home to be $\mathcal{F}_{\overline{Family}}(\textit{see-location})$. By synchronizing with the actions of the social network application registering the arriving and leaving actions of the user (enter(l) and leave(l) respectively), the policy can be represented by a finite state machine as in Figure 4. For simplicity, we use $\mathcal{F}_w(r)$ and $\mathcal{F}_{\overline{f}}(l)$, to represent $\mathcal{F}_{co-workers}(\textit{read-post})$ and $\mathcal{F}_{\overline{Family}}(\textit{see-location})$ respectively.

File Format Policy: An example from [60] considers a scenario where an application that writes characters from the set {a, b, c} to a file, through multiple write operations, and policy specifying that *"At the end of the sequence of writes, the file's content must respect a specific format where each string should end with a special character {!, ?}", which cannot occur elsewhere in the string.*. Hence the input alphabet is Σ = {a,b,c,!, ?}. The property can be specified by an LTL formula as follows:

$$(\text{a} \vee \text{b} \vee \text{c})\ \mathbf{U}\ ((\text{!} \vee \text{?}) \rightarrow \mathbf{G}\bot)$$

where $\mathbf{G}\bot$ indicates that no more input actions can be accepted. The policy can also be specified using a finite state machine as shown in figure 5. The initial state is S_0 and the only accepting state is S_3.

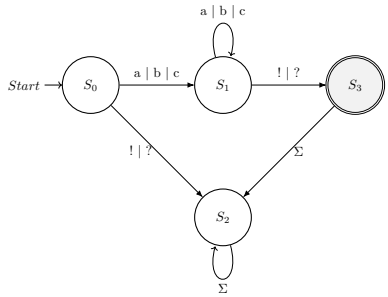


Figure 5: A finite-state machine representing the property that *each string should end by a special character* ! *or* ?" [60].

4 Altering Input Traces

In the situation where an input trace of event violates the policy, one may consider the possibility of modifying this trace so that it becomes compliant: this is the problem of runtime enforcement. In order to do so, we must first formalize how these modifications are applied. In this section, we introduce the notion of *proxy*, which is a transducer allowing to turn an input trace of events into one or more "modified" versions. We then enumerate particular categories of proxies corresponding to enforcement mechanisms from past literature, and describe a few possible notations to define such proxies.

4.1 Proxies

Formally, a *proxy* is a 1-monotonic transducer $\pi : \Sigma^* \to (2^{\Sigma^*})^*$. It takes as input a trace of events, and outputs a sequence of sets of traces $S_1, S_2, \ldots$. We add the soundness condition that for every $i > 0$ and every $\overline{\sigma} \in S_i$, there exists a $\overline{\sigma}' \in S_{i-1}$ such that $\overline{\sigma}' \preceq \overline{\sigma}$.

Intuitively, the proxy can be seen as a device that ingests an input sequence of events, and outputs after each step a set of sequences S_i; this set corresponds to the sequences that the proxy "suggests" in replacement of the first i events of the original input trace. In this context, the soundness condition is easier to understand: it corresponds to the fact that a proxy is allowed to extend, possibly in more than one way, any trace that was present in its previous set (including the empty trace), however it is not allowed to take back a trace that was proposed previously. Since a trace produced by a proxy is actually made of multiple traces, we call it a *multi-trace*.

This model can be illustrated graphically in the form of a *prefix tree* such as the ones depicted in Figure 6, where nodes of the tree are labelled with events. Each path from the root of the tree to one of its leaves represents one possible trace. The proxy starts from a tree made of a single root node; upon receiving each input event, it has the freedom to append any number of nodes to any node of the current tree. This is represented by the sequence in the figure, where additions with respect to the previous step are highlighted.

The sequence corresponds to the sets of traces $\{a, b\}$, $\{aa, a, b\}$, and $\{aa, a, ab, ba, bbb\}$, respectively. A few observations must be made on this sequence. First, note that it follows the soundness condition expressed above, in that every sequence in a set is the prefix of some sequence in the set that comes after. Also note that the ϵ node, representing the empty event, needs to be added to indicate that both a trace and one of its suffixes are present in the set (as is the case for a and aa). Finally, one can also observe that more than one event can be appended to an existing trace at once, as is illustrated in the bottom left of the last prefix tree.

A proxy π is said to be *combinatorial* if for every $\overline{\sigma} \in \Sigma^*$ such that $|\overline{\sigma}| = n$, there exists a sequence of sets $S_i \subseteq \Sigma^*$ for $i \in [1, n]$ such that $\pi(\overline{\sigma})[n] = S_1 \cdot \ldots \cdot S_n$. At each step of its operation, a combinatorial proxy chooses a set of trace fragments S_i, which are taken as the possible extensions of any trace in its current set. The proxy is called "combinatorial", as the traces it defines are the concatenation of any combination of fragments picked from each S_i.

When described in terms of prefix trees, a combinatorial proxy is such that at any given step, it adds the same set of nodes to all the leaves of the current tree. For instance, the proxy of Figure 6 is *not* combinatorial, as at each step, existing traces are not all extended in the same way. In contrast, Figure 7 shows the operation of a combinatorial proxy. The dashed lines delineate the portions of the prefix tree that are successively appended. Note how at each step, an identical node structure is appended to every leaf of the previous step. At the bottom of the figure, the sets of fragments $S_1, \ldots, S_n$ corresponding to each structure are represented. One can check that every

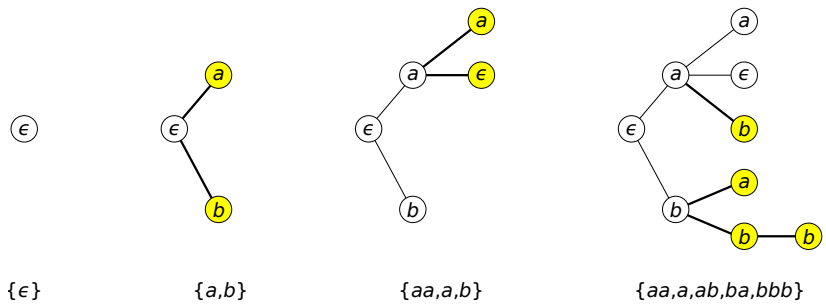


Figure 6: A sequence of prefix trees depicting the output of an arbitrary proxy, where each addition with respect to the previous tree is highlighted. At the bottom are written the explicit traces corresponding to each prefix tree.

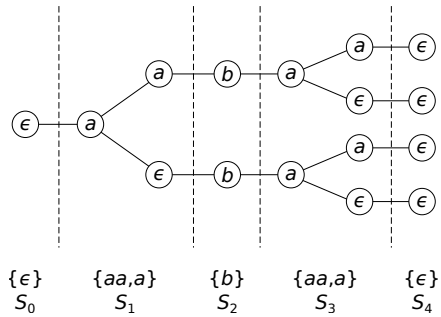


Figure 7: Illustration of the operation of a combinatorial proxy.

path in the tree is indeed the concatenation of a combination of a fragment taken from each of the S_i.

Combinatorial proxies can be seen as independent on output history, in that the possible ways in which an output trace is extended are the same regardless of the content of that trace. Hence, in Figure 7, the set S_1 indicates that any possible trace produced so far can be extended either by one or by two a events. Although they can produce a strict subset of all behaviours available to an unrestricted proxy, in practice almost every enforcement mechanism presented in past literature can be abstracted in the form of a combinatorial proxy, suggesting that this notion captures an important feature of enforcement.

In addition, many combinatorial proxies admit a natural representation under the form of an input-output automaton, such as a Mealy machine [57]. We recall that a Mealy machine is a variant of a finite-state automaton where transitions are labelled with an input symbol from an alphabet Σ_I, and an output symbol from another alphabet Σ_O.

A Mealy machine can be defined as a 6-tuple $\langle S, S_0, \Sigma_I, \Sigma_O, T, G\rangle$ where: S is a finite set of states, S_0 is the initial state, Σ_I is a finite set of input alphabet, Σ_O is a finite set of output alphabet, $T : S \times \Sigma_I \to S$ is a transition function mapping pairs of a state and an input symbol to the corresponding next state, and $G : S \times \Sigma_I \to \Sigma_O$ is an output function mapping pairs of a state and an input symbol to the corresponding output symbol. The transition and output functions can be coalesced into a single function $T : S \times \Sigma_I \to S \times \Sigma_O$.

For a given event alphabet Σ, a combinatorial proxy can be specified as a special case of a Mealy machine where $\Sigma_I = \Sigma$, and $\Sigma_O = 2^{\Sigma^*}$. The input symbol corresponds to the input event given to the proxy, and the output "symbol" of the corresponding transition is the set of fragments that extend each possible trace (i.e. one of the S_i in the definition above). Figure 8 gives an example of such a Mealy machine. We use $*$ on a transition to indicate any event not mentioned in another outgoing transition from the same state, and the notation $*/\{*\}$ to indicate that an event should be output as is. Intuitively, this proxy outputs all input events without modification, except sequences of b where any b after the first may or may not be deleted from the output (represented by the fact that b and ϵ are the two possible extensions of a trace in that case).

4.2 Categories of proxies

The proxy is a high-level abstraction of any security mechanism that modifies the underlying execution to ensure compliance with the desired security policy. As such, it can best be seen as a component

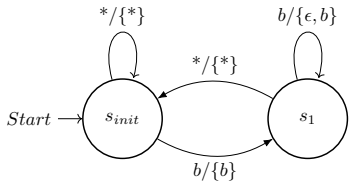


Figure 8: A proxy suppressing successive b events following an initial b.

of an Enforcement Monitor (EM), as defined in by Erlingsson [29]. Our previous definition is generic, and gives the proxy almost unlimited freedom to modify the input trace it receives in various ways —even by outputting traces that are completely unrelated to the input. In practice however, past literature has concentrated on specific types of proxies with stricter bounds on the modifications they are allowed to apply to a trace.

As detailed in section 2.1, the main aspect that distinguishes different classes of EM monitors is their capacity to alter the input event stream. Early work on run-time enforcement usually distinguished between (1) EM that is only capable of aborting (truncating) the execution; (2) EM that can insert addition events in the input stream, (3) EM can suppress (delete) events from the input stream and (4) EM that are able to both insert and delete events [10]. This latter type is said to have the capacity to *edit* the input sequence.

Each of these alternatives corresponds to a different implementation strategy. For example, aspect oriented programming [52] can be used to insert code segments that are executed when different point-cuts are encountered. This allows the implementation of an insertion proxy. Conversely, a firewall or an IDS interposed between the user and a host system operates as a suppression monitor, since it can prevent service requests from being executed (in effect suppressing them). The heightened capabilities of an edition EM require the use of a more involved program-rewriting method.

As observed by Erlingsson [29], the monitor (termed reference monitor or RM) can also be inserted at different layers of the architecture, with a consequent impact on its ability to affect the execution. Erlingsson distinguishes three cases, illustrated in Figure 9. First, the monitor may operate inside the operating system's kernel space, and prevent the execution of sensitive instructions (left). Alternatively, an untrusted program may be run in an interpreter, with simulates the execution and interposes itself between the program and the operating system (center). Finally, the monitor may be inlined inside of the target program, through a rewriting or code injection process (right). The execution of a program modified in such a manner can then the thought of as equivalent to the simultaneous execution of the program and the reference monitor.

4.3 Examples

We now refer to the policies we specified in section 3.3 and talk briefly about the enforcement mechanism used to enforce each policy:

For the Next and HasNext pattern Policy, the policy is violated whenever a $Next$ event appears without being preceded by a $HasNext$ event. To enforce the policy, possible actions can be done: inserting a $HasNext$ event, suppressing any $Next$ until a $HasNext$ appears event or maintaining a buffer and storing all the $Next$ events until a $HasNext$ appears then output all the events in the buffer. As an example, Figure 10 shows an enforcement monitor that enforces the policy by suppressing any $Next$ event coming before a $HasNext$ event.

The mechanism used to enforce the fair dispatcher ordering policy [30] aims to reorder events in a specific order. To do this, several buffers are used to store events temporarily to be later used in their correct order. To avoid buffering of large number of events, a buffer purging technique to delete an event from the buffer, and healing technique is used to add an event into the trace that may correct the order without buffering the current event. Once reaching a specified threshold K_{heal} (indicating the number of events in buffer after which healing is allowed) or K_{purge} (indicating the number of events in buffer after which purging is allowed), healing or purging are used respectively.

To enforce the privacy policies in Online Social Networks [58], two tools are used to communicate between each other: An OSN with a built-in enforcement for static privacy policies, and the LARVA tool [21] to monitor the evolution of the OSN and control the state of the policies at each moment in time.

For the file format policy enforcement, Pinisetty *et al.* [60] consider the case of a predictive runtime enforcement assume that the system is not

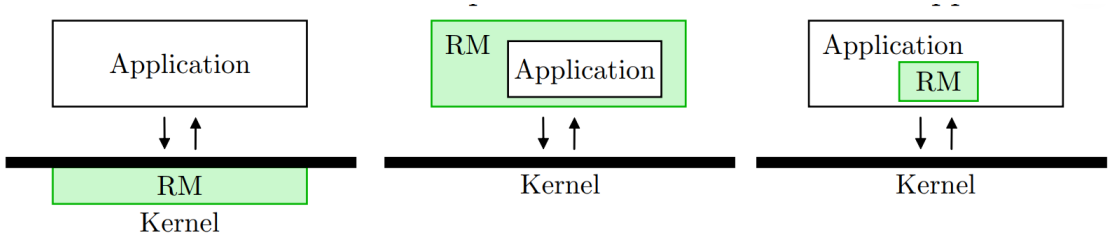


Figure 9: The implementation of a monitor, from [29].

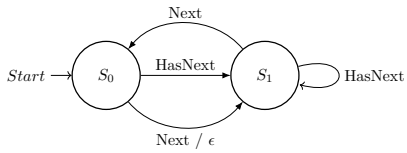


Figure 10: A graphical representation of an enforcement monitor of the Next and HasNext pattern policy shown in Figure 2.

entirely black-box, but they know something about its behaviour based on prediction. The *a priori* knowledge of the system allows the EM to emit an output event instantly rather than delaying them until more events arrive, or even permanently blocking them.

5 Producing Corrected Traces

The previous two sections have presented the notions of policy and proxy in isolation. On one side, policies were expressed regardless of how they could concretely be enforced; on the other side, proxies were defined as entities that could alter an input trace, but without any specific aim. In this section, we leverage these two concepts, and describe a mechanism through which involves an interplay between the output of a monitor for a given policy, and the alterations a proxy can inflict on an input trace, in order to produce a trace that guarantees compliance with the policy (or more precisely, non-violation).

5.1 A Definition of Correction

A proxy can be distinguished in whether it produces traces that are compliant or not with respect to a policy Φ. Formally, a proxy π is called *strongly* Φ*-preserving* if for any given input $\overline{\sigma} \in \Sigma^*$ such that $\pi(\overline{\sigma}) = \Sigma_1^*, \Sigma_2^*, \ldots, \Sigma_n^*$, any $i \in [1, n]$ and any $\overline{\sigma}' \in \Sigma_i^*$, we have that $\varphi(\overline{\sigma}') \in \{\top, \top^?\}$. The proxy strongly preserves Φ if it only outputs sequences that result in a positive verdict of the monitor induced by Φ. Similarly, π is *weakly* Φ*-preserving* if the latter condition is replaced by $\varphi(\overline{\sigma}') \in \{\top, \top^?, \bot_?\}$. The proxy weakly preserves Φ if it only outputs sequences that do not result in a definitive negative verdict with respect to Φ.

At first glance, it would seem to be sufficient to merely pipe an event trace into a Φ-preserving proxy π, and pick any of its output traces as the corrected one. This solution discards two important elements. First, assuming that π is Φ-preserving is a strong hypothesis, which couples the possible actions of a proxy with a specific policy. This means that a new proxy must be designed for every policy (or even every change of a policy). In addition, even ensuring that a given π is Φ-preserving is a nontrivial task.

The second element to consider is the notion of transparency touched upon earlier. We recall that our definition of proxy can, in theory, correct an input trace in ways that are completely unrelated to the input, even for traces that are already in compliance with the policy. One must therefore ensure, at least, that prefixes of the input that do not violate the policy should be left as they are. But this still leaves imprecise at what point a portion of the input should be replaced by a corrected version, and by how much. It also does not stipulate if the proxy should keep on correcting the input forever after this moment, or if the contents of the input trace could be used again after some time, and from what point.

In the following, we clarify these questions by proposing a formal definition of how a proxy π is expected to interact with an input trace, with respect to an independently-specified policy Φ. We do so by considering an "original" input trace $\overline{\sigma} \in \Sigma^*$ and a "corrected" output trace $\overline{\sigma}' \in \Sigma^*$, and

give conditions on how these two traces should be related.

Given a trace $\overline{\sigma}$ and a suffix $\overline{\sigma}'$ to be appended to it, we say that $\overline{\sigma}'$ is *positive throughout* if $\varphi(\overline{\sigma} \cdot \overline{\sigma}'') \in \{\top, \top^?\}$ for every $\overline{\sigma}'' \preceq \overline{\sigma}'$. Intuitively, the suffix is positive throughout if appending each event successively always results in the monitor producing a positive verdict. We say that $\overline{\sigma'}$ *starts negatively* if $\varphi(\overline{\sigma} \cdot \overline{\sigma}'[0]) \in \{\bot_?, \bot\}$. The suffix starts negatively if appending its first event to $\overline{\sigma}$ produces a negative verdict. Finally, the suffix *ends positively* if $\varphi(\overline{\sigma} \cdot \overline{\sigma}') \in \{\top, \top^?\}$ –that is, if the monitor's verdict after appending all its events is positive.

We divide $\overline{\sigma}$ and $\overline{\sigma}'$ into n *segments* such that $\overline{\sigma} = \overline{\sigma}_1 \cdot \ldots \cdot \overline{\sigma}_n$, and $\overline{\sigma}' = \overline{\sigma}'_1 \cdot \ldots \cdot \overline{\sigma}'_n$. The corresponding segments of each trace do not need to be of same length. We call such a division of the original and corrected trace a *segmentation*. We designate by *corrected prefix up to* i the prefix of the corrected trace containing the first i segments (this prefix being set to ϵ for $i = 0$). A segmentation of $\overline{\sigma}$ and $\overline{\sigma}'$ is called a *correction* if, for every corrected prefix up to i ($i \in [0, n]$), $\overline{\sigma}_{i+1}$ is positive throughout and $\overline{\sigma}'_{i+1} = \overline{\sigma}_{i+1}$, or $\overline{\sigma}_{i+1}$ starts negatively and $\overline{\sigma}'_{i+1}$ ends positively.

Figure 11 shows an example of such a correction. The squares at the top represent the events of the original trace, and those at the bottom the events of the corrected trace. Each of these events is grouped according to a possible segmentation, with the boundaries of the corresponding segments in the original and corrected trace linked by dashed lines. The color and symbol inside an event of a segment σ_i correspond to the verdict the monitor reaches when ingesting the corrected segment up to $i-1$, plus the events in the segment up to this point. For instance, the verdict associated to the event x in the figure is obtained by feeding the monitor the first two segments of the corrected trace, and then event x of the original trace, i.e. $\varphi(\overline{\sigma}'_1 \cdot \overline{\sigma}'_2 \cdot x)$.

The diagram shows the various situations covered by the definition. Segment $\overline{\sigma}_1$ is positive throughout; therefore segment $\overline{\sigma}'_1$ is identical. This corresponds to the transparency requirement, which imposes that a prefix of the original trace that satisfies the policy should be left untouched. Segment $\overline{\sigma}_2$ starts negatively. This gives the signal that this event and whatever comes after can be substituted by another segment, $\overline{\sigma}'_2$. In the example, observe that the corrected segment contains more events than the input segment it is matched with. Also observe that, as per the definitions above, the corrected segment must end positively. Thus, a transducer producing a corrected trace can replace a segment of arbitrary length that starts negatively by another segment, but this new segment must be such that the resulting trace satisfies the policy. That is, it cannot emit a correction that places the trace on a "cliffhanger" where it still does not satisfy the policy. However, note how $\overline{\sigma}'_2$ contains events resulting in a negative verdict. The only requirement is that the policy be satisfied at the end of the prefix.

On its side, $\overline{\sigma}_3$ is positive throughout, and thus must be output identically as $\overline{\sigma}'_3$. This is an important aspect of our definition of correction: once a segment has been corrected and the trace be put into a satisfying state, events of the input trace from this point on must resume being output without modification as long as the trace satisfies the policy. This condition is stricter than existing definitions of transparency, which only impose that a prefix of the original trace should be let through until the first violating event. That is, once the input trace violates the policy, the classical definition of enforcement allows a proxy to alter the trace, and does not rule out that it can do so forever. In contrast, our definition of correction stipulates that, once a violating input segment has been replaced by a fixed version, control must be relinquished to the original input trace until a new violation is found.

5.2 Selecting Corrections

Referring back to the definition of *enforcement* as defined in [10], we see that a monitor is considered to have enforced a property if it can transform an invalid sequence into *any* valid sequence. A transparency requirement is present in the definition, but it only limits the transformations that the monitor may perform on valid traces. Being consistent with this definition, the monitor may be able to enforce a property, but not in a way that is necessarily useful or desirable. Indeed, few people would accept a security mechanism that 'corrects' a misbehaving execution by replacing it with a benign input that is completely unrelated to the original execution.

Consequently, additional constraints must be imposed on the behaviour of the monitor in order

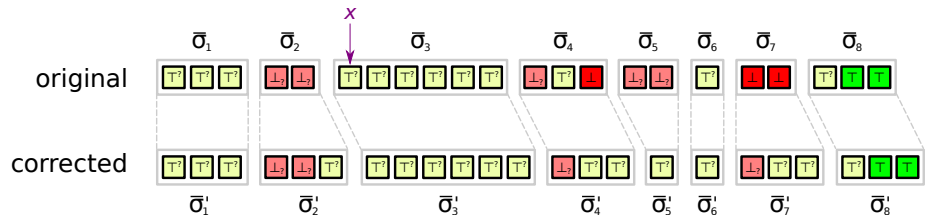


Figure 11: Illustration of the relationship between an input trace and a possible correction.

to ensure meaningful enforcement. For example, Bielova *et al.* [14] define a series of monitors whose output is syntactically, related to input sequence — for instance imposing that the corrected sequence always be a prefix of the input sequence. Alternatively, Khoury *et al.* [49] suggest that all possible sequences be arranged in a preorder, and that the monitor be required to select a solution that falls higher that the input on this preorder. In this paper, we adopt a similar, but more flexible solution: by separating the generation of potential solutions from the selection of the optimal (or preferred) solutions, we do away with the complexity of creating such a preorder and ensuring that the monitor behaves in a manner that is conformant with this added restriction.

A *scoring function* is a function $\rho : \Sigma^* \to \mathbb{R}$, which assigns to each sequence in Σ^* a real value called its *score*. This function induces a total ordering $\sqsubseteq$ on traces, such that $\overline{\sigma} \sqsubseteq \overline{\sigma}'$ if and only if $\rho(\overline{\sigma}) \leq \rho(\overline{\sigma}')$. In principle, any preorder can be used to select the optimal corrected sequence. In practice, some preorders can be exhibit properties that may be desirable. First, a preorder can be *monotonic*, meaning that a corrective action taken by the monitor can never be made irremediably 'wreck' a sequence, in the sense that a continuation of original sequence falls higher on the preorder that any possible corrected sequence. Formally: $\forall \overline{\tau}, \overline{\tau}' \in \Sigma^* : \neg\exists \overline{\sigma} \in \Sigma^\omega : \forall \overline{\sigma}' \in \Sigma^\omega : \overline{\tau} \sqsubseteq \overline{\tau}' \wedge \overline{\tau}' \cdot \sigma' \sqsubseteq \overline{\tau} \cdot \overline{\sigma}$.

Secondly, a preorder can be *truth-correlated*, meaning that any valid trace has a score higher than any invalid trace. Formally, this means that for a given policy φ we have that $\forall \overline{\tau}, \overline{\tau}' \in \Sigma^\omega : \varphi(\overline{\tau}) \to \varphi(\overline{\tau}') \Rightarrow \overline{\tau}' \sqsupseteq \overline{\tau}$.

A natural example of choice for the preferred trace could be one that minimizes the number of modifications (insertions and deletions) on the input. This can be captured by turning a given input alphabet Σ into an alternate alphabet $\widehat{\Sigma}$, where each symbol $\sigma \in \Sigma$ exists in three versions: σ designates an event of an output trace that was already present in the input, $\sigma_\downarrow$ designates an event that was added to the output, and $\sigma^\uparrow$ designates an event that was deleted from the input. For example, the trace abc, to which b is deleted and a is inserted at the end, would result in the trace $ab^\uparrow ca_\downarrow$. Evaluating a policy on such a trace can be done by simply handling any $\sigma_\downarrow$ as σ, and any $\sigma^\uparrow$ as ϵ. Thus the previous trace would be handled by a policy (and its associated monitor) in the same way as aca.

Equipped with this notation, defining a scoring function that correlates with modifications is straightforward: for a given trace $\overline{\sigma}$, one simply adds -1 for each occurrence of an inserted or deleted event (starting from 0). Thus a higher-ranking value corresponds to a corrected trace with fewer modifications. However, this is by far not the only possible ranking one can build. As another example, one could imagine a function that assigns a higher score to a sequence that satisfies the policy for as many prefixes as possible. Concretely, this could be defined as $\rho(\overline{\sigma}) \triangleq |\{\overline{\sigma}' \preceq \overline{\sigma} : \mu(\overline{\sigma}') \in \{\top^?, \top\}\}|$. The experiments in Section 7 will describe other scoring functions specific to some use cases.

6 A Modular Runtime Enforcement Pipeline

Equipped with the notions of monitor, proxy and the concept of corrected trace, we now 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.

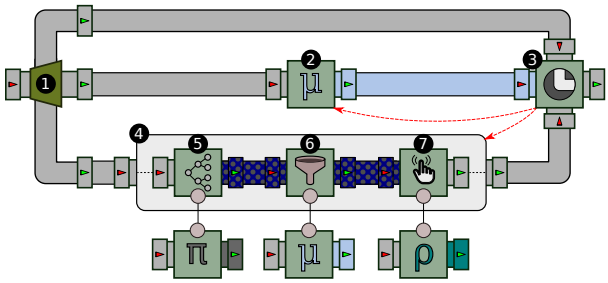


Figure 12: The stages of the runtime enforcement framework. Events flow from left to right.

6.1 Pipeline Description

The proposed enforcement model takes the form of a "pipeline", which is a chain of transducers taking as its input a possibly incorrect event trace, and producing as its output a sequence of events that satisfies the definition of correction with respect to a policy introduced in Section 5.1. The high-level schematics of the model are shown in Figure 12. Various transducers are represented as boxes illustrated with different pictograms, depending on their definition. These transducers are organized along a data flow graph where events move from left to right. A link between two transducers indicates that the output of the first is given as the input to the second.

The diagram uses different colors to represent events of different types, which will be explained later. The pipeline is parameterized by three transducers, labelled μ, π and ρ. First, μ is a *monitor* responsible for evaluating a security policy on a trace. Transducer π is the *proxy*, which is tasked with applying modifications to an input trace. Finally, ρ is the *ranking* transducer, and assigns a numerical score to a trace based on an enforcement preorder.

The global operation of the pipeline can be summarized as follows. 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 the monitor μ (box #2). 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 π (#5); this set of sequences is then filtered out so that only sequences satisfying the security policy evaluated by μ are kept (#6). The last phase sends the remaining sequences into the ranking transducer ρ, 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 and end with a discussion of the advantages of this model.

6.1.1 Production of Corrected Traces

In our earlier definition of a proxy, each output multi-event contains the complete set of sequences proposed in replacement of the input. This introduces a large amount of repetition, since a prefix of each of these sequences was already present in the set produced for the previous event. It also prevents the output of the proxy to be ingested by another downstream transducer in a progressive manner. Better yet would be a representation which, upon each input event, only produces a description of what is appended to the existing prefix tree. To this end, we introduce a further restriction on the sequence of prefix trees induced by a proxy, by imposing that each leaf of a given tree be appended by at least one node in the next tree, and that only single nodes can be added at each step (instead of sequences of nodes).

One can reason that a proxy following this constraint can produce the same set of sequences as an unconstrained one; Figure 13 illustrates this. It shows a sequence of prefix trees following the restriction added in this section. It can be observed that the final prefix tree represents the same set of traces as that of Figure 6. However, this is obtained at the price of additional ϵ nodes, due to the condition that each leaf must be appended with at least one node at every step. In addition, one more step in the sequence is required to append the two b events at the bottom of the tree that were added in a single step in Figure 6.

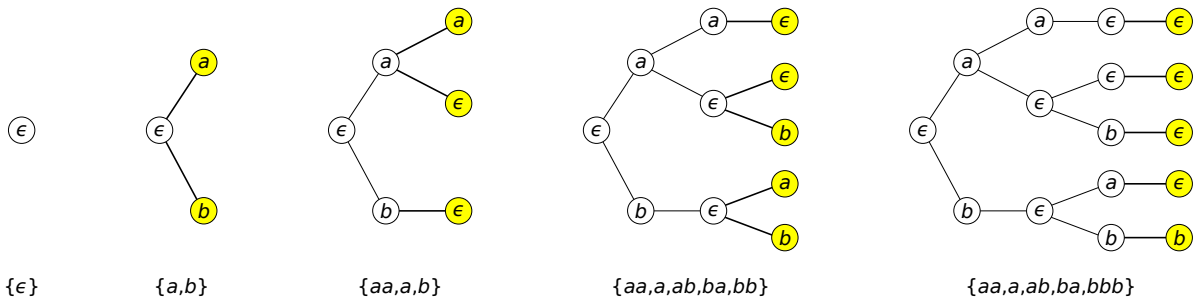


Figure 13: A modified sequence of prefix trees where only leaves are modified, and where a single event at a time is appended to each branch. This sequence ends up producing the same set of traces as that of Figure 6.

The restriction adds some complexity to the trees; in counterpart, they give a regular structure to these trees where each branch has the same length, and where each further step adds a single layer of nodes to the leaves of the current tree. In counterpart, this regularity can be exploited; 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 $v_0 = \langle[\,]\rangle$, 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 sequence in Figure 13 corresponds to the sequence of prefix tree elements:

$$\langle[\,]\rangle \quad (1)$$
$$\langle[a], [b]\rangle \quad (2)$$
$$\langle[\epsilon], [\epsilon, b], [a, b]\rangle \quad (3)$$
$$\langle[\epsilon], [\epsilon], [\epsilon], [\epsilon], [b]\rangle \quad (4)$$

Note that a symbol may be ϵ, 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 12 is precisely to receive each sequence set produced from π, and turn it into the appropriate prefix tree element.

6.1.2 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.

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 13, 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 μ on each path of the induced prefix tree, and to remove a node (as well as all its descendants) as soon as μ 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 μ_{σ_i} represent the state of monitor μ 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 μ_{σ_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 μ_{σ_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.

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.

6.1.3 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 the ranking transducer $\rho : \Sigma^* \to \mathbb{R}$, 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 ρ, and the path that maximizes the score is selected and returned as the output.

The operation of the selector, depicted in Figure 12 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 ρ_{σ_i} and the score s_i this transducer has produced after processing σ_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 σ_i can be chosen based on the highest score s_i in all pairs.

Algorithm 1 Incremental filtering

```
procedure FILTER([μ_σ1, ..., μ_σn], [v_0, ..., v_n])
  v ← [ ], m ← [ ]
  for i ← 1, n do
    v' ← [ ]
    for x ∈ v_i do
      ADD(m, μ_σi·x)
      if μ_σi(x) contains ⊥ then
        ADD(v', ◊)
      else if i = n and μ_σi(x) ends with ⊥^?
        ADD(v', ◊) else ADD(v, v')
  return (m, v)
end procedure
```

Algorithm 2 Output trace selection

```
1:  procedure UPDATE([(ρ_σ1, s_1), ..., (ρ_σn, s_n)],
    [v_0, ..., v_n])
2:    m ← [ ]
3:    for i ← 1, n do
4:      for x ∈ v_i do
5:        s = −∞
6:        if x ≠ ◊ then
7:          s ← LAST(ρ_σ1(x))
8:        ADD(m, ρ_σi·x)
9:    return m
10: end procedure
```

6.1.4 Merging Valid vs. Corrected Traces

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 μ 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 μ does not produce the verdict $\bot$ or $\bot^?$ 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 μ 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 μ 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" μ 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 12, and which we call *checkpointing*. Along with the transducer μ of box #2, a copy μ_σ is kept of that transducer, in the state it was after reading σ (the "checkpoint"). Intuitively, σ represents the sequence of events that have actually been output by the pipeline. As events are received, μ updates its internal state accordingly, but μ_σ is preserved. This copy is updated only when the downstream gate instructs it to, by providing a segment of newly output events σ'. When this occurs, both the checkpoint μ_σ and the internal state of μ 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.

6.1.5 Event Buffering

A final aspect of the architecture that needs to be discussed is the notion of buffering. The default behaviour 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 definition of a corrected trace in Section 5.1 and the proposed architecture both 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. This greedy strategy does not guarantee the absolute best course of action, unless the scoring transducer is *suffix-monotonic*. Formally, a transducer τ is suffix-monotonic if for every triplet of sequences σ_1, σ_2 and σ_3, the fact that $\tau(\sigma_1) \leq \tau(\sigma_2)$ implies that $\tau(\sigma_1 \cdot \sigma_3) \leq \tau(\sigma_2 \cdot \sigma_3)$. In such a case, one can easily see that picking the best choice at every event guarantees the best score overall.

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 ϵ 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. Obviously, the appropriate choice is specific to the use case and the nature of the properties involved in the enforcement pipeline.

The pipeline as defined ensures transparency as it is usually defined. A more conservative strategy would be to only alter the execution if the input sequence irremediably violates the security policy. This strategy guarantees compliance with transparency as it is usually defined, but limits the monitor to the enforcement of safety properties, since a violation of a liveness property can always be remediated by subsequent actions. However, one may instead opt for a more flexible notion of transparency, which allows modifications of valid traces, as long as the output is guaranteed to be higher than the input on the enforcement preorder. In many cases, the strategy employed will be context specific, imposing that some element of the input be preserved, or obligating the selector to take action once a specific event is encountered in the input trace. This would likely be the case for most transactional properties.

Regardless of the strategy chosen, one should keep in mind that the notion of "optimal" output sequence must be qualified with respect to the choices available to the selector at the moment an output must be produced.

6.2 Use Cases

We now illustrate the operation of this pipeline by describing two use cases, on which we completely define all elements of the workflow.

6.2.1 Use Case 1: Museum

In the first use case, we look into an example taken from Drabik *et al.* [27], which considers two sorts of visitors entering to a museum: children and adults, and guards responsible about protecting the visitors.

We are interested in a more complex scenario by studying the behaviour of the principals while both entering and leaving the museum. The set of events that can occur in the trace of a museum: g^+, c^+ and a^+ indicating a guard, a child and an adult entering the museum respectively, and g^-, c^- and a^- indicating a guard, a child and an adult leaving the museum respectively.

Monitor

To keep the children safe while they are inside the museum, a policy stating that *access is forbidden for any child if no guard is currently present in the museum* should be enforced. Adults are allowed to enter on their own, however children must be accompanied by a guard. The property involves keeping track of the number of children inside the museum as it increases and decreases over c^+ and c^- events that may occur, as well as the number of guards that changes over the g^+ and g^- events. 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 [24]. 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

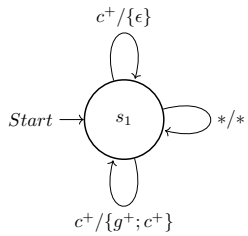


Figure 14: Representation of a possible proxy enforcing the museum use case.

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.

$$t_1 := \text{ite}(g^+; t_1[-1,0] + 1; \text{ite}(g^-; t_1[-1,0] - 1; t_1[-1,0]))$$
$$t_2 := \text{ite}(c^+; t_2[-1,0] + 1; \text{ite}(c^-; t_2[-1,0] - 1; t_2[-1,0]))$$
$$\varphi := \text{ite}(\varphi[-1,\top], (t_1 - t_2) \geq 0, \bot)$$

The first equation defines a stream that keeps the count of the number of guards inside the museum. This counter is incremented by one whenever a guard enters to the museum, decremented by one whenever a guard leaves the museum, and keeps its value otherwise. The second equation does the same thing for children. We assume that $t_1 = 0$ and $t_2 = 0$ whenever an expression refers to a position before the start of the stream. The third equation represents the idea that the number of children inside the museum should never exceed the number of guards. A Boolean stream defined as φ, whose output can be used as the monitor verdict for the security policy. The equation is made such as the property remains false once it becomes false.

Proxy

The interest of this scenario lies in the possible variations for the proxy and enforcement preorder. The policy can be enforced by refusing (suppressing) the entrance of a child when the number of guards inside in the museum is zero, or by lending (inserting) a guard when needed.

One may also buffer all c^+ events until a c^- event appears then we can output one c^+ event from the buffer, or until a g^+ event appears then we can output all the c^+ events from the buffer.

In Figure 14, we provide an example of a proxy for the museum example defined as a Mealy machine. This proxy may either suppress a c^+ or insert a g^+ event.

Selector

This policy exposes itself to several interrelated courses of actions, with the choices made by the monitor restricting its future course of action: refusing a high number of children incurs its own trade-off, since the museum will lose an amount of profit that it may gain if the children are allowed to enter. Similarly, adding more guards also incurs a trade-off because the museum must afford the salaries paid to these guards. 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 [48] for complete details.

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 $\widehat{\mathcal{C}}^v_\varphi$, where φ 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 φ evaluates to v. We write $\widehat{\mathcal{C}}^{\geq ?}_\varphi$ as a stand-in for $\widehat{\mathcal{C}}^?_\varphi + \widehat{\mathcal{C}}^\top_\varphi$ and $\widehat{\mathcal{C}}^{\leq ?}_\varphi$ for $\widehat{\mathcal{C}}^?_\varphi + \widehat{\mathcal{C}}^\bot_\varphi$. The values returned by counters range over $\mathbb{N}$, but arithmetic operators or functions can be freely applied to the outputs of multiple counters over the same sequence to compute information about the trace, yielding a value in $\mathbb{R}$.

Aside from $\widehat{\mathcal{C}}$, TK-LTL defines a number of other counters. $\mathcal{C}$ counts the number of prefixes that satisfy a given property, the unary $\mathcal{D}^v_\varphi$ counter returns the initial point in the input trace where a given property holds, the binary counter ${}_\phi\mathcal{D}^v_\varphi$ the first position at which a condition holds, starting from the satisfaction of another condition while the counter $\mathcal{L}^v_\varphi$ returns the index of the last occurrence of an event for which the property φ evaluates to v. The semantics of these counters, along with usage examples, are given in [48]. In addition to counters, the semantics of TK-LTL includes quantifiers. One of them is Quantifiers examine the value returned by a counter for each prefix of the input sequence, and return a value from the same 3-valued truth domain as an LTL-property according to a condition subscripted to the quantifier. TK-LTL defines three quantifiers: the first two are the existential and universal quantifiers, with natural semantics. For instance, the formula $\exists_{=5}\widehat{\mathcal{C}}^\top_p$ returns $\top$ if the

atomic proposition p holds on at least 5 prefixes of the input trace, and returns '?' otherwise. Conversely, the formula $\exists_{<0}\widehat{\mathcal{C}}_p^\top - \widehat{\mathcal{C}}_q^\top$ returns $\top$ iff there exists a prefix of the input trace for which the atomic proposition q holds more often than p. the propositional quantifier, and is written as $\mathcal{P}$. The formula $\mathcal{P}_{\sim k}\widehat{\mathcal{C}}$ thus evaluates to $\top$ if the comparison $n \sim k$ holds where n is the value returned by $\widehat{\mathcal{C}}$. For example, let $\sigma = aaaba$ be a trace; the formula $\mathcal{P}_{=3}\widehat{\mathcal{C}}_a^\top$ evaluates to $\top$ at positions $i = 3$ and $i = 4$, and to $\bot$ elsewhere.

Quantifiers, such as $\forall_{\sim k}\widehat{\mathcal{C}}$ or $\exists_{\sim k}\widehat{\mathcal{C}}$, verify if the values returned by a counter meet a given condition c, and return a verdict in V. This allows unlimited recursion of alternating LTL formulae, counters and quantifiers.

The possible enforcement preorders in the museum use case can be to minimize the number of modifications to the trace, 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).

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 $\widehat{\mathcal{C}}_{g^+}^\top$ counts the total number guards that enter the museum, and the TK-LTL subformula $\widehat{\mathcal{C}}_{g^-}^\top$ counts the total number guards that leaves the museum. Hence, the formula $\widehat{\mathcal{C}}_{g^+}^\top - \widehat{\mathcal{C}}_{g^-}^\top$ can be used as a transparency constraint if the museum's main concern is to minimize the total number of 'idle' guards that are inside the museum. A monitor that seeks to achieve this goal will thus avoid inserting g^+ events in the input stream.

Similarly, the formula $\widehat{\mathcal{C}}_{c^+}^\top - \widehat{\mathcal{C}}_{c^-}^\top$ expresses an alternative transparency requirement, namely maximizing gains for the museum by allowing the highest number of visitors to enter including the children.

6.2.2 Use Case 2: Casino

As a more complex example, we now consider a variant of the scenario from Colombo *et al.* [22], which stems from a study of the remedial actions that can be taken to recover from violations of the terms of smart contracts. Since smart contract are transactional in nature and cannot be modified after they are deployed, the framework proposed in this paper is especially well suited to this situation.

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 pre-specified 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 (players who wish to bet more simply output multiple bet events), and the $Pay()$ event transfers a single dollar. We omit from events any element of parameter value that does not bear consequence on the discussion of the event at hand. For instance, it is safe to assume that the $EndGame()$ action indicates the id of the game that must be ended, but we need not concern ourselves with these implementation details. 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. This property involves keeping track of the dealer's balance as it increases and decreases over Pay events that may occur.

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.

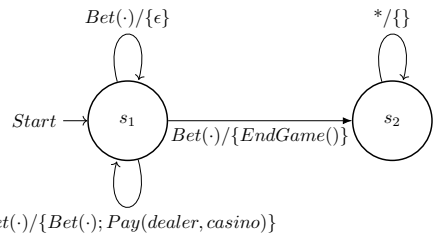


Figure 15: Representation of a possible proxy enforcing the casino use case.

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

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. It increments by one when a player pays the casino, and decrements by that same amount in the reverse situation. Otherwise, the balance is left unchanged. We assume that $t_1 = 0$ and $t_2 = k$ whenever an expression refers to a position before the start of the stream, where k represents the dealer's initial balance. The third equation represents the idea that the potential payouts should never exceed the current balance. A Boolean stream defined as φ, whose output can be used as the monitor verdict for the security policy. The equation is made such as the property remains false once it becomes false.

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. Several formalisms can be used to represent the proxy. Furthermore, since the proxy is stated independently of the monitor and the selector, a different formalism can be used to represent each. In Figure 15, we provide an example of a proxy for the casino example defined as a Mealy machine. This proxy may either suppress a bet or terminate the game (by inserting an $EndGame()$ event).

Selector

The pipeline will choose among several courses of actions to attain one of several goals: cancelling 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.

As in the museum use case, the enforcement preorder and most of the possible requirements can be formulated using simple formulas in TK-LTL. The subformula $\widehat{\mathcal{C}}^{\top}_{Bet(\cdot)}$ 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 $\widehat{\mathcal{C}}^{\top}_{Pay(casino,\cdot)} - \widehat{\mathcal{C}}^{\top}_{Pay(\cdot,casino)}$ expresses an alternative transparency requirement, namely maximizing gains for the casino.

Two other properties that could be applied for this use case. The first is maximizing the number of games ran simultaneously, which can expressed by the formula $\widehat{\mathcal{C}}^{\top}_{NewGame(\cdot)} - \widehat{\mathcal{C}}^{\top}_{EndGame(\cdot)}$; the second is minimizing the number of bets that are placed while no games are running, and can be expressed as:

$$\widehat{\mathcal{C}}^{\top}_{\mathcal{P}_{=0}\widehat{\mathcal{C}}^{\top}_{NewGame(\cdot)} - \widehat{\mathcal{C}}^{\top}_{EndGame(\cdot)} \wedge Bet(\cdot)}$$

Other goals could be possible, such as maximizing the number of different players that participate in a game, or minimizing the number of games with no or few bettors.

7 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

Transparency Requirement	TK-LTL Formula
Maximize gains to the Casino	$\widehat{\mathcal{C}}^{\top}_{Pay(casino,\cdot)} - \widehat{\mathcal{C}}^{\top}_{Pay(\cdot,casino)}$
Maximize the total number of bets that are placed	$\widehat{\mathcal{C}}^{\top}_{Bet(\cdot)}$
Highest number of games ran simultaneously	$\widehat{\mathcal{C}}^{\top}_{NewGame(\cdot)} - \widehat{\mathcal{C}}^{\top}_{EndGame(\cdot)}$
Minimize the number of bets that are placed while no games are running	$\widehat{\mathcal{C}}^{\top}_{\mathcal{P}_{=0}\widehat{\mathcal{C}}^{\top}_{NewGame(\cdot)} - \widehat{\mathcal{C}}^{\top}_{EndGame(\cdot)} \wedge Bet(\cdot)}$

Table 1: Representation of four possible transparency constraints using TK-LTL.

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 [40].

7.1 Overview of BeepBeep

BeepBeep [36, 40] is a tool that can perform various tasks over event streams of different nature. The fundamental building block of BeepBeep is called *processor*. A processor takes one or more event streams as its input, performs a computation on the elements of these streams and returns one or more event streams as its output. Several commonly used functionalities are already present across a number of palettes represented as libraries (i.e. JAR files), and the user can define new processors or functions to be used with BeepBeep's core elements.

BeepBeep has few features that distinguish it from other event processing systems, such as being *intuitive* in the sense that any computation done in a processor can be expressed in a graphical way using a set of pictograms as in Figure 12. A processor object is represented by a square box, with a pictogram that indicates the type of computation it executes on events. On the sides of this box are one or more "pipes" representing its inputs and outputs. Another feature is being *lightweight* with a stand-alone Java library that weighs less than 250 kilobytes and low memory requirements.

A third feature is it having a *modular* architecture in which all of its functionalities are packed into palettes, which the user can include into their project only if they need its contents. This is in contrast to many other systems that seek to deliver a massive, one-size-fits-all set of functionalities. Customized computations are possible over event traces by allowing processors to be composed; this means that the output of a processor can be redirected to the input of another, creating complex processor chains. Events can either be *pushed* through the inputs of a chain, or *pulled* from the outputs, and BeepBeep takes care of managing implicit input and output event queues for each processor. In addition, users also have the freedom of creating their own custom processors and functions, by extending the Processor and Function objects, respectively.

Extensions of BeepBeep with predefined custom objects are represented into palettes; there exist palettes for various purposes, such as signal processing, XML manipulation, plotting, and finite-state machines. BeepBeep has been used in a variety of case studies [17, 38, 51, 63, 69].

7.2 Implementation

The pipeline described in Section 6.1 has been implemented as a stand-alone BeepBeep extension. 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 μ, π and ρ representing the monitor, proxy and ranking transducer described earlier.

In line with the formal presentation of Section 6, 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 [37] makes it possible to specify the monitor using finite-state machines, LTL, Lola, or Quantified Event Automaton [62], 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 6.1.5. 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 behaviour implementing another strategy. In the experiments described later, 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 12. To the end user, this processor can be used as a box receiving a sequence of events in Σ and producing another sequence of events in Σ, 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.

Figure 16 shows a concrete example of how such a pipeline can be instantiated. First, a processor `mu` corresponding to the monitor is created. In the code example, this monitor is taken to be a Moore machine, whose states and transitions would be defined through a series of calls to a method named `addTransition` (a single example of which is shown in the excerpt). The next instruction instantiates the processor that is to act as the proxy `pi`; in this case, a processor already provided by our extension is used, called `InsertAny`. It is a predefined proxy that can, upon any input event, insert before it a fixed number of other events. The precise way in which it is instantiated in the example makes it such that either event a or event b may be inserted before any input event.

The next instruction instantiates the processor acting as the ranking transducer `rho`. This time again, a predefined processor is used (`CountModifications`), which increments the score of an input trace by 1 for every event that is either added or deleted. Finally, a `Gate` processor encompassing the pipeline of Figure 12 is created by passing as parameters the processors defined earlier. The presence of `IntervalFilter` is the processor that implements the strategy deciding on when to output a corrected segment. In this case, it is instructed to wait for at most 1 input event before producing a correction.

The remaining lines show how, once instantiated, this `Gate` can be used like any other BeepBeep processor. That is, the gate is connected to an event sink, and events are then pushed to its input in standard BeepBeep fashion. Assuming that the policy implemented by monitor `mu` corresponds to the condition "no two successive a must be present", the contents of the sink after pushing a twice is the trace aba. It is consequent with the fact that: 1. the original input trace violates the policy; 2. the proxy is allowed to insert b anywhere in the input; 3. the filter is instructed to wait for at most 1 input event before issuing a corrected version; 4. the trace aba is indeed a corrected trace that complies with the policy.

7.3 Scenarios

As one can see, a few lines of code suffice to create an enforcement pipeline where each parameter can be an arbitrary chain of BeepBeep processors. This makes the implementation an excellent playground to experiment with various policies and proxies. Therefore, 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 [39], which is a self-contained executable package containing all the code required to rerun them [67]. For each variation of a scenario, we ran the enforcement pipeline on a randomly generated trace of length 1,000

```
// Define the monitor verifying the policy
Processor mu = new StateMooreMachine(1, 1);
mu.addTransition(0, new EventTransition("a", 1));
...

// Define the proxy
Processor pi = new InsertAny(1, "a", "b");

// Define the selector
Processor rho = new CountModifications();

// Instantiate the pipeline with
Gate g = new Gate(mu, pi,
  new IntervalFilter(pi, 1), rho);

// Connect the gate to a sink and push events
QueueSink s = new QueueSink();
Connector.connect(g, s);
Pushable p = g.getPushableInput();
p.push("a");
p.push("a");
```

Figure 16: Code usage of the runtime enforcement pipeline.

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. 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.

In addition to the *Museum* and *Casino* use cases 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 Life cycle

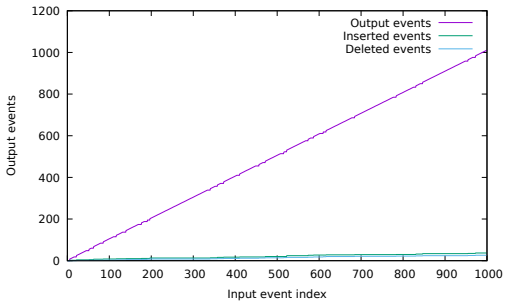
The second scenario is related to the operations that can be made on a resource such as a file, and is a staple of runtime verification literature [20]. A trace of events is made of interleaved operations open, close, read and write on multiple files. The policy is notable in that it is *parametric*: it splits the trace into multiple sub-traces (one for each file), and stipulates that each file follows a prescribed life cycle (read and write are allowed only between open and close, and no write can occur after a read). The monitor for this policy is a Moore machine embedded into a BeepBeep `Slice` processor. The scenario reuses a proxy and ranking transducer from *Simple*.

7.4 Impact on Overhead

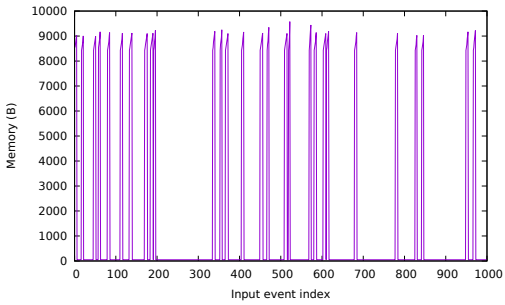
A first important measurement is the impact of the use of the runtime enforcement pipeline on the running time and memory consumption of the system.

The results on this aspect are summarized in Table 2. 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 [12]. 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



(a) Input vs. output events



(b) Memory consumption

Figure 17: Runtime statistics for the execution of an enforcement pipeline on a variation of the museum scenario.

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.

The action of a proxy can also be examined in further detail. Figure 17a 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 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.

On its side, Figure 17b 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 17a 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 6.

7.5 Proxy Comparison

An interesting side effect of the proposed implementation is that it makes it relatively easy to compare the effect of various enforcement strategies and scoring functions for the same policy and the same input sequence. To this end, it suffices to create a different instance of the `Gate` processor and varying some of its input parameters. This section discusses such a comparison, by focusing on the *Museum* scenario described earlier.

We consider four different enforcement strategies:

1. *Children shadow*: in this case, the proxy is allowed to insert a guard before every child entering the museum. It consequently takes a guard out of the museum every time a child gets out (that is, each guard "shadows" a child). Any other guard can come in, but is prevented from going out. Other events are left unmodified. This proxy is notable for being memory-less: it is not required to keep any information from the past to perform its actions.
2. *Delete children*: the proxy keeps an exact count of children and guards. It deletes any c^+ event when no guard is in the museum, and inserts as many c^- events as there are children in the museum when the last guard gets out. In other words, this proxy prevents from entering or throws children out, depending on the presence of guards.
3. *Insert guard*: as with the previous proxy, this one counts children and guards. It inserts a g^+ whenever a child enters a museum with no guard inside, or when the last guard gets out and children are still in the museum. Otherwise, the input events are let through without modification.

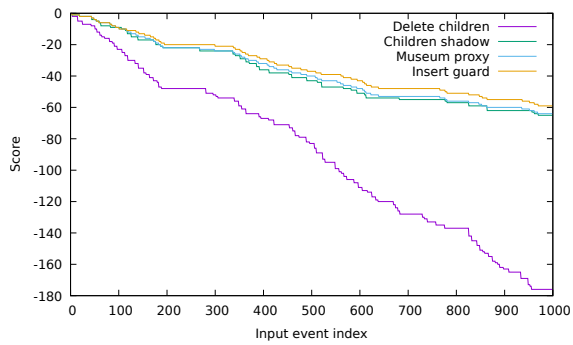
Event source	Policy	Proxy	Scoring formula	Throughput	Max memory
Casino	Casino policy	Casino proxy	Maximize bets	2380	9824
			Maximize gains	490	7976
			Minimize changes	2325	8814
Files	All files lifecycle	Delete any	Minimize changes	78	9580
Museum	Museum policy	Museum proxy	Maximize children	4347	9580
			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 2: Summary of throughput (in events/sec.) and maximum memory consumption (in bytes) for each scenario.

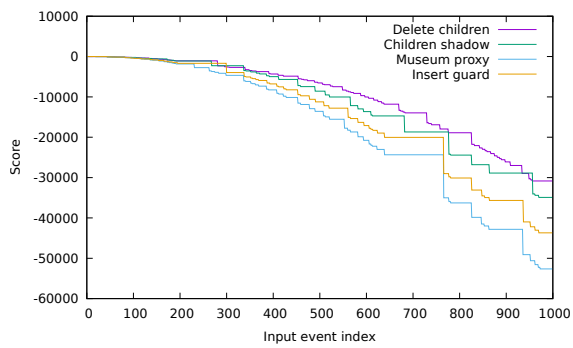
4. *Museum proxy*: this is the proxy used in the experiments of the previous section. It has more freedom than the previous ones: if a child enters, the proxy may first insert a "guard in" event or delete the 'child in" event. If a guard exits, the proxy may delete the "guard out" event. Contrary to the previous ones, this proxy makes these modifications to the trace without any regard to the state of the policy. As per the definition of our enforcement pipeline, it is up to the downstream selector to weed out corrections made by this proxy that are not compliant with the policy. This means this proxy could not be used as classical enforcement monitor, and must be encased in our proposed enforcement pipeline.

These proxies are tested against the same input trace, and their respective impact on the input trace is empirically measured by looking at the amount of modifications each incurs on that input. The results are presented in Figure 18a. As explained in Section 5.2, the ranking function in such a case starts from 0 and subtracts 1 for every added or deleted event in the selected output trace. Thus traces are assigned a negative score, with a higher value indicating fewer modifications.

As one can see in the figure, the effect of each proxy on the trace results in different scores depending on the enforcement strategy. The *Delete children* strategy, in particular, introduces substantially more changes to the input trace than the remaining ones. This is expected, as when the last guard comes out, all children currently in the museum are expelled at once, resulting in a potentially large number of c^- being inserted into the



(a) Minimize changes



(b) Minimize idle guards

Figure 18: The comparison of the action of different proxies on the same input sequence, for two ranking functions.

corrected trace. In contrast, other proxies exhibit a less invasive (and ultimately roughly equivalent) behaviour on the input trace.

Note however that this impact depends on the proxy, but also on the enforcement preorder. This is exemplified in Figure 18b, which trades function ρ for a new version where each trace starts with a score of 0, and is decremented by 1 for each time step where guards are in the museum without any children. Note how this preorder is uncorrelated to the amount of changes being made to the input: a large number of modifications will be deemed preferable if it ensures a smaller number of idle guards in the museum. According to this metric, this time the proxies are reversed. In this case, *Delete children* turns out to be the proxy producing higher-scoring corrections than the others.

7.6 Discussion

This proposed model 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" [66]. The difference between the two models lies in the fact that a multi-event contains multiple single events, while a sequence set contains multiple event sequences.

The pipeline proposed in Figure 12 should be contrasted with the classical enforcement monitors (EM) considered in past literature, which take the form of Figure 1. In an EM, an input sequence is transformed into a corrected output sequence in a single step. It is up to the EM to keep track of the specification's current state, decide on the appropriate modifications to apply to each incoming input event (including possibly buffering this event and decide later), and produce a *single* output trace which must be guaranteed to satisfy the policy. As we have seen, automatic synthesis algorithms for such enforcement monitors are rare, entailing that they must typically be designed by hand for each policy and set of available corrective actions. Alas, this task turns out to be nontrivial even for simple cases, and formally proving that an EM always produces a valid output regardless of its input is equally challenging.

In addition, our definition of correction, introduced in Section 5.1, is different from what it typically expected of an EM. To illustrate this, consider the enforcement monitor illustrated in Figure 19. Given the input trace $a^+a^+c^+c^+$,

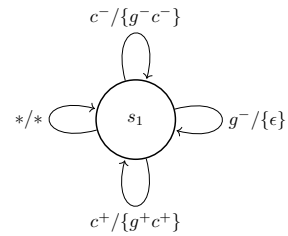


Figure 19: An enforcement monitor applying the *Children shadow* correction strategy in the *Museum* example.

which violates the museum policy (no child in without a guard), it produces the output trace $a^+a^+g^+c^+g^+c^+$. One can check that this sequence indeed satisfies the policy, and moreover that the prefix of the input that satisfies the policy (a^+a^+) has been output without modification (as required by the basic transparency expectation).

However, this enforcement monitor does *not* produce an output sequence that satisfies our definition of correction. After ingesting the first three events, the EM produces the sequence $a^+a^+g^+c^+$, inserting the g^+ event required to restore satisfaction of the policy. But then, since this corrected output places the policy back to a valid state, the next input event (c^+) does not introduce a violation. Following the terminology of Section 5.1, it represents a segment that is positive throughout, and thus must be output without modification. Thus, to abide by the definition of correction, one cannot simply add a guard before every child.[1] This simple example illustrates that our proposed definition of corrected trace is tighter than existing requirements on an enforcement monitor, and narrows the amount of intervention that one is allowed to make on the input.

It is also important to stress that in our proposed pipeline, 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

[1] When used as a proxy in our enforcement pipeline, the Mealy machine of Figure 19 does not create this issue, as on the second c^+ input event, control is not switched to the enforcement pipeline as no violation is detected.

of the proxy is to generate all the possible modifications of the input trace that are potentially available to enforce a given property.

This generic definition presents a few advantages. First, it is agnostic to the actual representation of the enforcement capabilities. Figure 8 shows an example of a proxy that applies a suppression modification action; given the input trace $\sigma = babbc$, it produces the output $\{b\}, \{a\}, \{b\}, \{\epsilon, b\}, \{c\}$. An interesting feature of this model is to enable "non-standard" enforcement capabilities. For instance, classical delete automaton 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 (illustrated by the Mealy machine of Figure 8). 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.

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 behaviour of the enforcement monitor need not be coded explicitly. Instead, the behaviour 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 6, 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. As with previous phases, the model leaves open the question of how ρ is specified. In principle, any formal model could be used to state the transparency requirement. For the enforcement preorder, several formalism could potentially be used, including Lola [24], fuzzy-time Linear Temporal Logic [35], or TK-LTL[48], an extension of LTL.

Some examples, taken from the literature, illustrate the flexibility of the approach. Recall the "no send after read" policy introduced in Section 2.1. As discussed above, the policy can be expressed by inserting an entry in the log, by suppressing the send event, by suppressing the read event or by aborting the execution. In this case, the property would be enforced by assigning a value to each trace, based on the behaviour in presents (a truncated trace being naturally less valuable than a longer trace). 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.

8 Conclusion

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 behaviour 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 7 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.

An important contribution of this paper is that it uses several categories of proxies from the literature with various enforcement strategies and scoring functions and empirically compares the effectiveness of these proxies when used with the same policy and input sequence. The paper also presents a notion of correction of traces, compares the enforcement pipeline to enforcement using an automata model that strictly enforces the policy, and shows that this notion narrows the amount of intervention that proxy is allowed to make on an input trace compared to an enforcement monitor.

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 12 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 [66]. 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.

Conflict of Interests

Not Applicable.

References

[1] Ancona D, Dagnino F, Franceschini L (2018) A formalism for specification of java API interfaces. In: Dolby J, Halfond WGJ, Mishra A (eds) Companion Proceedings for the ISSTA/ECOOP 2018 Workshops, ISSTA 2018, Amsterdam, Netherlands, July 16-21, 2018. ACM, pp 24–26, https://doi.org/10.1145/3236454.3236476, URL https://doi.org/10.1145/3236454.3236476

[2] Avalle M, Pironti A, Sisto R (2014) Formal verification of security protocol implementations: A survey. Form Asp Comput 26(1):99–123. https://doi.org/10.1007/s00165-012-0269-9, URL https://doi.org/10.1007/s00165-012-0269-9

[3] Bartocci E, Majumdar R (eds) (2015) Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings, Lecture Notes in Computer Science, vol 9333, Springer, https://doi.org/10.1007/978-3-319-23820-3, URL https://doi.org/10.1007/978-3-319-23820-3

[4] Bartocci E, Falcone Y, Francalanza A, et al (2018) Introduction to runtime verification. In: Bartocci E, Falcone Y (eds) Lectures on Runtime Verification - Introductory and Advanced Topics, Lecture Notes in Computer Science, vol 10457. Springer, p 1–33, https://doi.org/

10.1007/978-3-319-75632-5_1, URL https://doi.org/10.1007/978-3-319-75632-5_1

[5] Bauer A, Jürjens J (2010) Runtime verification of cryptographic protocols. Computers & Security 29(3):315–330. https://doi.org/https://doi.org/10.1016/j.cose.2009.09.003, URL https://www.sciencedirect.com/science/article/pii/S0167404809001047, special issue on software engineering for secure systems

[6] Bauer A, Leucker M, Schallhart C (2006) Monitoring of real-time properties. In: Arun-Kumar S, Garg N (eds) FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science, 26th International Conference, Kolkata, India, December 13-15, 2006, Proceedings, Lecture Notes in Computer Science, vol 4337. Springer, pp 260–272, https://doi.org/10.1007/11944836_25, URL https://doi.org/10.1007/11944836_25

[7] Bauer A, Leucker M, Schallhart C (2007) The good, the bad, and the ugly, but how ugly is ugly? In: Sokolsky O, Tasiran S (eds) Runtime Verification, 7th International Workshop, RV 2007, Vancouver, Canada, March 13, 2007, Revised Selected Papers, Lecture Notes in Computer Science, vol 4839. Springer, pp 126–138, https://doi.org/10.1007/978-3-540-77395-5_11, URL https://doi.org/10.1007/978-3-540-77395-5_11

[8] Bauer A, Leucker M, Schallhart C (2010) Comparing LTL semantics for runtime verification. J Log Comput 20(3):651–674

[9] Bauer A, Leucker M, Schallhart C (2011) Runtime verification for LTL and TLTL. ACM Trans Softw Eng Methodol 20(4):14:1–14:64. https://doi.org/10.1145/2000799.2000800, URL https://doi.org/10.1145/2000799.2000800

[10] Bauer L, Ligatti J, Walker D (2002) More enforceable security policies. In: In Foundations of Computer Security

[11] Beauquier D, Cohen J, Lanotte R (2013) Security policies enforcement using finite and pushdown edit automata. Int J Inf Sec 12(4):319–336

[12] Betti Q, Montreuil B, Khoury R, et al (2020) Smart Contracts-Enabled Simulation for Hyperconnected Logistics, Springer International Publishing, Cham, pp 109–149

[13] Bielova N, Massacci F (2011) Do you really mean what you actually enforced? Int J of Inf Security pp 1–16. URL http://dx.doi.org/10.1007/s10207-011-0137-2, 10.1007/s10207-011-0137-2

[14] Bielova N, Massacci F (2011) Do you really mean what you actually enforced? - edited automata revisited. Int J Inf Sec 10(4):239–254

[15] Bielova N, Massacci F (2012) Iterative enforcement by suppression: Towards practical enforcement theories. Journal of Computer Security 20(1)

[16] Bodden E, Hendren LJ, Lam P, et al (2010) Collaborative runtime verification with tracematches. J Log Comput 20(3):707–723. https://doi.org/10.1093/logcom/exn077, URL https://doi.org/10.1093/logcom/exn077

[17] Boussaha MR, Khoury R, Hallé S (2017) Monitoring of security properties using beepbeep. In: Imine A, Fernandez JM, Marion J, et al (eds) Foundations and Practice of Security - 10th International Symposium, FPS 2017, Nancy, France, October 23-25, 2017, Revised Selected Papers, Lecture Notes in Computer Science, vol 10723. Springer, pp 160–169, https://doi.org/10.1007/978-3-319-75650-9_11, URL https://doi.org/10.1007/978-3-319-75650-9_11

[18] Chabot H, Khoury R, Tawbi N (2011) Extending the enforcement power of truncation monitors using static analysis. pp 194–207

[19] Chang E, Manna Z, Pnueli A (1993) The safety-progress classification. In: Bauer FL, Brauer W, Schwichtenberg H (eds) Logic and Algebra of Specification. Springer Berlin Heidelberg, Berlin, Heidelberg, pp 143–202

[20] Chen F, Meredith PO, Jin D, et al (2009) Efficient formalism-independent monitoring of parametric properties. In: ASE. IEEE

Computer Society, pp 383–394

[21] Colombo C, Pace GJ, Schneider G (2009) LARVA — safer monitoring of real-time java programs (tool paper). In: Hung DV, Krishnan P (eds) Seventh IEEE International Conference on Software Engineering and Formal Methods, SEFM 2009, Hanoi, Vietnam, 23-27 November 2009. IEEE Computer Society, pp 33–37, https://doi.org/10.1109/SEFM.2009.13, URL https://doi.org/10.1109/SEFM.2009.13

[22] Colombo C, Ellul J, Pace GJ (2018) Contracts over smart contracts: Recovering from violations dynamically. In: Margaria T, Steffen B (eds) ISoLA 2018, LNCS, vol 11247. Springer, pp 300–315

[23] Convent L, Hungerecker S, Leucker M, et al (2018) Tessla: Temporal stream-based specification language. In: Massoni T, Mousavi MR (eds) Formal Methods: Foundations and Applications - 21st Brazilian Symposium, SBMF 2018, Salvador, Brazil, November 26-30, 2018, Proceedings, Lecture Notes in Computer Science, vol 11254. Springer, pp 144–162, https://doi.org/10.1007/978-3-030-03044-5_10, URL https://doi.org/10.1007/978-3-030-03044-5_10

[24] D'Angelo B, Sankaranarayanan S, Sánchez C, et al (2005) LOLA: runtime monitoring of synchronous systems. In: TIME. IEEE Computer Society, pp 166–174

[25] Dolzhenko E, Ligatti J, Reddy S (2015) Modeling runtime enforcement with mandatory results automata. Int J Inf Sec 14(1):47–60. https://doi.org/10.1007/s10207-014-0239-8, URL https://doi.org/10.1007/s10207-014-0239-8

[26] Dolzhenko E, Ligatti J, Reddy S (2015) Modeling runtime enforcement with mandatory results automata. Int J Inf Secur 14(1):47–60

[27] Drábik P, Martinelli F, Morisset C (2012) Cost-aware runtime enforcement of security policies. In: Jøsang A, Samarati P, Petrocchi M (eds) STM, LNCS, vol 7783. Springer, pp 1–16

[28] Drábik P, Martinelli F, Morisset C (2012) A quantitative approach for inexact enforcement of security policies. In: Gollmann D, Freiling FC (eds) ISC 2012, LNCS, vol 7483. Springer, pp 306–321

[29] Erlingsson U (2004) The inlined reference monitor approach to security policy enforcement. PhD thesis, USA, aAI3114521

[30] Falcone Y, Salaün G (2021) Runtime enforcement with reordering, healing, and suppression. In: Calinescu R, Pasareanu CS (eds) Software Engineering and Formal Methods - 19th International Conference, SEFM 2021, Virtual Event, December 6-10, 2021, Proceedings, Lecture Notes in Computer Science, vol 13085. Springer, pp 47–65, https://doi.org/10.1007/978-3-030-92124-8_3, URL https://doi.org/10.1007/978-3-030-92124-8_3

[31] Falcone Y, Mounier L, Fernandez JC, et al (2011) Runtime enforcement monitors: Composition, synthesis, and enforcement abilities. Form Methods Syst Des 38(3):223–262

[32] Falcone Y, Fernandez J, Mounier L (2012) What can you verify and enforce at runtime? Int J Softw Tools Technol Transf 14(3):349–382. https://doi.org/10.1007/s10009-011-0196-8, URL https://doi.org/10.1007/s10009-011-0196-8

[33] Falcone Y, Mariani L, Rollet A, et al (2018) Runtime failure prevention and reaction. In: Bartocci E, Falcone Y (eds) Lectures on Runtime Verification - Introductory and Advanced Topics, LNCS, vol 10457. Springer, p 103–134

[34] Fong PWL (2004) Access control by tracking shallow execution history. In: S&P 2004. IEEE Computer Society, pp 43–55

[35] Frigeri A, Pasquale L, Spoletini P (2014) Fuzzy time in linear temporal logic. ACM Trans Comput Log 15(4):30:1–30:22

[36] Hallé S, Khoury R (2017) Event stream processing with beepbeep 3. In: Reger G,

Havelund K (eds) RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools, September 15, 2017, Seattle, WA, USA, Kalpa Publications in Computing, vol 3. EasyChair, pp 81–88, https://doi.org/10.29007/4cth, URL https://doi.org/10.29007/4cth

[37] Hallé S, Khoury R (2018) Writing domain-specific languages for BeepBeep. In: Colombo C, Leucker M (eds) RV, LNCS, vol 11237. Springer, pp 447–457, https://doi.org/10.1007/978-3-030-03769-7_27

[38] Hallé S, Gaboury S, Bouchard B (2016) Activity recognition through complex event processing: First findings. In: Bouchard B, Giroux S, Bouzouane A, et al (eds) Artificial Intelligence Applied to Assistive Technologies and Smart Environments, Papers from the 2016 AAAI Workshop, Phoenix, Arizona, USA, February 12, 2016, AAAI Technical Report, vol WS-16-01. AAAI Press, URL http://www.aaai.org/ocs/index.php/WS/AAAIW16/paper/view/12561

[39] Hallé S, Khoury R, Awesso M (2018) Streamlining the inclusion of computer experiments in a research paper. Computer 51(11):78–89. https://doi.org/10.1109/MC.2018.2876075

[40] Hallé S (2018) Event Stream Processing With BeepBeep 3: Log Crunching and Analysis Made Easy. Presses de l'Université du Québec

[41] Hallé S, Villemaire R (2008) Runtime monitoring of message-based workflows with data. pp 63–72, https://doi.org/10.1109/EDOC.2008.32

[42] Hamlen KW, Morrisett JG, Schneider FB (2006) Computability classes for enforcement mechanisms. ACM Trans Program Lang Syst 28(1):175–205

[43] Hamlen KW, Morrisett JG, Schneider FB (2006) Computability classes for enforcement mechanisms. ACM Trans Program Lang Syst 28(1):175–205. https://doi.org/10.1145/1111596.1111601, URL https://doi.org/10.1145/1111596.1111601

[44] Havelund K, Goldberg A (2008) Verify Your Runs, vol 4171, pp 374–383. https://doi.org/10.1007/978-3-540-69149-5_40

[45] Jaksic S, Bartocci E, Grosu R, et al (2018) Quantitative monitoring of STL with edit distance. Formal Methods Syst Des 53(1):83–112. https://doi.org/10.1007/s10703-018-0319-x, URL https://doi.org/10.1007/s10703-018-0319-x

[46] Khoury R, Hallé S (2015) Runtime enforcement with partial control. In: García-Alfaro J, Kranakis E, Bonfante G (eds) FPS 2015, LNCS, vol 9482. Springer, pp 102–116

[47] Khoury R, Hallé S (2015) Runtime enforcement with partial control. In: García-Alfaro J, Kranakis E, Bonfante G (eds) Foundations and Practice of Security - 8th International Symposium, FPS 2015, Clermont-Ferrand, France, October 26-28, 2015, Revised Selected Papers, Lecture Notes in Computer Science, vol 9482. Springer, pp 102–116, https://doi.org/10.1007/978-3-319-30303-1_7, URL https://doi.org/10.1007/978-3-319-30303-1_7

[48] Khoury R, Hallé S (2018) Tally keeping-LTL: An LTL semantics for quantitative evaluation of LTL specifications. In: IRI 2018. IEEE, pp 495–502

[49] Khoury R, Tawbi N (2012) Corrective enforcement: A new paradigm of security policy enforcement by monitors. ACM Transactions on Information and System Security 15(2):10

[50] Khoury R, Tawbi N (2012) Which security policies are enforceable by runtime monitors? a survey. Computer Science Review 6(1):27–45

[51] Khoury R, Hallé S, Waldmann O (2016) Execution trace analysis using LTL-FO ˆ+. In: Margaria T, Steffen B (eds) Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part II, pp 356–362, https://doi.org/10.1007/

978-3-319-47169-3_26, URL https://doi.org/10.1007/978-3-319-47169-3_26

[52] Kiczales G, Lamping J, Mendhekar A, et al (1997) Aspect-oriented programming. In: European conference on object-oriented programming, Springer, pp 220–242

[53] Koymans R (1990) Specifying real-time properties with metric temporal logic. Real Time Syst 2(4):255–299. https://doi.org/10.1007/BF01995674, URL https://doi.org/10.1007/BF01995674

[54] Legunsen O, Marinov D, Rosu G (2015) Evolution-aware monitoring-oriented programming. In: Bertolino A, Canfora G, Elbaum SG (eds) 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2. IEEE Computer Society, pp 615–618, https://doi.org/10.1109/ICSE.2015.206, URL https://doi.org/10.1109/ICSE.2015.206

[55] Leucker M, Schallhart C (2009) A brief account of runtime verification. J Log Algebraic Methods Program 78(5):293–303. https://doi.org/10.1016/j.jlap.2008.08.004, URL https://doi.org/10.1016/j.jlap.2008.08.004

[56] Manna Z, Pnueli A (1995) Temporal verification of reactive systems - safety. Springer

[57] Mealy GH (1955) A method for synthesizing sequential circuits. The Bell System Technical Journal 34(5):1045–1079. https://doi.org/10.1002/j.1538-7305.1955.tb03788.x

[58] Pace GJ, Pardo R, Schneider G (2016) On the runtime enforcement of evolving privacy policies in online social networks. In: Margaria T, Steffen B (eds) Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part II, pp 407–412, https://doi.org/10.1007/978-3-319-47169-3_33, URL https://doi.org/10.1007/978-3-319-47169-3_33

[59] Pinisetty S, Falcone Y, Jéron T, et al (2013) Runtime enforcement of timed properties. In: Qadeer S, Tasiran S (eds) Runtime Verification. Springer Berlin Heidelberg, Berlin, Heidelberg, pp 229–244

[60] Pinisetty S, Preoteasa V, Tripakis S, et al (2017) Predictive runtime enforcement. Formal Methods Syst Des 51(1):154–199. https://doi.org/10.1007/s10703-017-0271-1, URL https://doi.org/10.1007/s10703-017-0271-1

[61] Pnueli A (1977) The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977. IEEE Computer Society, pp 46–57, https://doi.org/10.1109/SFCS.1977.32, URL https://doi.org/10.1109/SFCS.1977.32

[62] Reger G, Cruz HC, Rydeheard DE (2015) MarQ: Monitoring at runtime with QEA. In: Baier C, Tinelli C (eds) TACAS 2015, LNCS, vol 9035. Springer, pp 596–610

[63] Roudjane M, Rebaine D, Khoury R, et al (2019) Predictive analytics for event stream processing. In: 23rd IEEE International Enterprise Distributed Object Computing Conference, EDOC 2019, Paris, France, October 28-31, 2019. IEEE, pp 171–182, https://doi.org/10.1109/EDOC.2019.00029, URL https://doi.org/10.1109/EDOC.2019.00029

[64] Schneider FB (2000) Enforceable security policies. ACM Trans Inf Syst Secur 3(1):30–50

[65] Selyunin K, Jaksic S, Nguyen T, et al (2017) Runtime monitoring with recovery of the sent communication protocol. In: Majumdar R, Kunčak V (eds) Computer Aided Verification. Springer International Publishing, Cham, pp 336–355

[66] Taleb R, Khoury R, Hallé S (2021) Runtime verification under access restrictions. In: Bliudze S, Gnesi S, Plat N, et al (eds) FormaliSE@ICSE 2021. IEEE, pp 31–41, https://doi.org/10.1109/FormaliSE52586.2021

[67] Taleb R, Hallé S, Khoury R (2022) Benchmark measuring the overhead of runtime enforcement using multi-traces (LabPal package). DOI: 10.5281/zenodo.5976001

[68] Talhi C, Tawbi N, Debbabi M (2006) Execution monitoring enforcement for limited-memory systems. In: PST. Association for Computing Machinery, New York, NY, USA, PST '06

[69] Varvaressos S, Lavoie K, Gaboury S, et al (2017) Automated bug finding in video games: A case study for runtime monitoring. Comput Entertain 15(1):1:1–1:28. https://doi.org/10.1145/2700529, URL https://doi.org/10.1145/2700529

[70] Vella M, Colombo C, Abela R, et al (2021) RV-TEE: secure cryptographic protocol execution based on runtime verification. J Comput Virol Hacking Tech 17:229–248

[71] Yu F, Bultan T, Ibarra OH (2011) Relational string verification using multi-track automata. Int J Found Comput Sci 22(8):1909–1924. https://doi.org/10.1142/S0129054111009112, URL https://doi.org/10.1142/S0129054111009112