Available online at www.sciencedirect.com

SciVerse ScienceDirect

journal homepage: www.elsevier.com/locate/cosrev

Survey

Which security policies are enforceable by runtime monitors? A survey

Raphaël Khoury [*,1], *Nadia Tawbi*

Département d'informatique et de génie logiciel, Université Laval, Québec, Canada

ARTICLE INFO

Article history:
Received 11 January 2011
Received in revised form
4 January 2012
Accepted 4 January 2012

Keywords:
Computer security
Runtime monitoring
Dynamic analysis
Security policies

ABSTRACT

Runtime monitoring is a widely used approach to ensure code safety. Several implementations of formal monitors have been proposed in the literature, and these differ with respect to the set of security policies that they are capable of enforcing. In this survey, we examine the evolution of knowledge regarding the issue of precisely which security policies monitors are capable of enforcing. We identify three stages in this evolution. In the first stage, we discuss initial limits on the set of enforceable properties and various ways in which this set can be extended. The second stage presents studies that identify constraints to the enforcement power of monitors. In the third stage, we present a final series of studies that suggest various alternative definitions of enforcement, which specify both the set of properties the monitors can enforce as well as the manner by which this enforcement is provided.

© 2012 Elsevier Inc. All rights reserved.

Contents

1. Introduction 28
2. Security policies and security properties 28
3. First stage: delineating the set of enforceable properties 30
   3.1. Security automaton modeling of monitors 30
   3.2. Extending the range of enforceable properties using the edit automata 30
   3.3. The edit automaton and infinite sequences 33
   3.4. Comparing mechanisms' enforcement power 34
4. Second stage: monitoring with memory and computability constraints 35
   4.1. Imposing computability constraints 35
   4.2. Monitoring and memory constraints 37
      4.2.1. Shallow history automata 37
      4.2.2. Bounded history automata 37
      4.2.3. Finite automaton 39

[*] *Corresponding author.* Tel.: +1 418 653 0177.
E-mail addresses: raphael.khoury.1@ulaval.ca (R. Khoury), nadia.tawbi@ift.ulaval.ca (N. Tawbi).

[1] Present address: Defence Research and Development Canada, Canadian Department of National Defence, Valcartier, G1W 3J7, Québec, Canada.

1574-0137/$ - see front matter © 2012 Elsevier Inc. All rights reserved.
doi:10.1016/j.cosrev.2012.01.001

5. Third stage: alternative definitions of enforcement ... 40
5.1. Defining subclasses to effective enforcement ... 40
5.2. Corrective enforcement ... 42
6. Conclusion ... 43
References ... 44

1. Introduction

As security concerns become increasingly central in our everyday interaction with complex systems, one solution that is rapidly gaining wide acceptance to address these concerns is run time monitoring, an approach to code safety that permits the execution of untrusted code by observing its execution and reacting as needed to avoid a violation of a security policy. This enforcement method has accordingly been the subject of a number of academic studies. Although these studies have explored a number of different topics related to the enforcement of security policies by monitors, one question seems to recur frequently in most if not all the works: exactly which set of properties are *monitorable*, in the sense that they are enforceable by monitors operating under different sets of constraints[2]? These are the policies for which a given monitor is capable of ensuring that a violation does not occur while operating under certain limiting constraints. This seemingly straightforward question does not admit a simple answer and has been the topic of numerous papers with conclusions that sometimes seem in conflict with each other. Yet determining in what context a given security policy becomes enforceable is central to the selection of the enforcement mechanism and even to the design of the security policy itself.

In this survey, we examine the various answers that have been proposed in the literature. As we will show, the set of properties enforceable by monitors is highly contingent on a number of factors, namely the availability of data about the target program's possible behavior, the means at the disposal of the monitor to react to a potential violation, memory and computability constraints and the definition of *enforcement* being used. A precise knowledge of these issues is essential to adequately select the best enforcement mechanism that can guarantee the respect of a given security policy. Furthermore, when the desired security policy falls outside the range of policies enforceable by the preferred enforcement mechanism, the research discussed in this survey can suggest ways to extend the mechanism's limits.

In our view, the evolution of knowledge with respect to the limits of the enforcement power of monitors can be subdivided into three stages of development. In the first stage, studies delineated the set of enforceable properties and suggested various ways this set could be extended. Researchers in this stage proposed increasingly powerful conceptual models of monitors culminating with the edit automaton, a very flexible model that can be used to capture the behavior of any runtime enforcement mechanism. Second stage studies, in contrast, dealt with constraints which limit the enforcement power of monitors. Research in this stage introduced memory and computability constraints into the models that had been previously proposed. Third stage studies take issue with the definition of enforcement used in the works of the previous two stages and propose alternative definitions to constrain the monitor's behavior further and capture restrictions on the monitors behavior when it is faced with a possible violation of the security policy. These latter studies ask not only *if* a given policy can be enforced by monitor, but also *how* this enforcement would take place, adding to the enforcement paradigm important limits on its permissible behavior, which are necessary to provide meaningful enforcement but where absent from previous models.

We begin in Section 2 by rigorously defining the notions of executions, security policy, and monitor that we will manipulate. In Section 3, we examine the first stage of studies, which show how these definitions allow us to state the limitations of a rudimentary monitor. From these limitations, the set of security properties enforceable by this monitor can be deduced. We then examine various ways to enhance the power of monitors by giving in each case a new formal definition of the extended monitor and identify the most promising model. We next turn to the enforcement power of those monitors found to be the most powerful and give a lower bound to the set of properties they can enforce. In Section 4 we examine the second stage of studies that focus on how computational and memory constraints can affect the set of enforceable properties. Finally, in Section 5, we revisit the notion of enforcement and propose alternative definitions put forth in the third stage of studies. Concluding remarks are given in Section 6.

2. Security policies and security properties

The first step in our analysis is to define a theoretical framework that can accommodate the various concepts we elaborate, such as executions, security policies, and monitors. To this end we adopt the formal notation devised in [3] which is widely used in the field.

An execution σ of a program, or trace, is modeled as a finite or infinite sequence of atomic program actions:

$$\sigma = a_0, a_1, a_2, \ldots.$$

We let a range over a finite or countably infinite set of atomic actions Σ. The empty sequence is noted ϵ, the set of all finite length sequences is noted Σ^*, that of all infinite length sequences is noted Σ^ω, and the set of all possible sequences

[2] In this study, we are uniquely interested with *runtime enforcement*, which refers to techniques that prevent violations of the security property. Monitors can additionally be used for runtime verification, which simply aims to detect the violation. An examination of the set of properties that are monitorable in the latter case falls outside the scope of this study. The interested reader is referred to [1,2].

is noted $\Sigma^\infty = \Sigma^\omega \cup \Sigma^*$. Likewise, for a set of sequences $\mathcal{S}$, $\mathcal{S}^*$ denotes the finite iterations of sequences of $\mathcal{S}$ and $\mathcal{S}^\omega$, that of infinite iterations, and $\mathcal{S}^\infty = \mathcal{S}^\omega \cup \mathcal{S}^*$. Let $\tau \in \Sigma^*$ and $\sigma \in \Sigma^\infty$ be two sequences of actions. We write $\tau; \sigma$ for the concatenation of τ and σ. We say that τ is a prefix of σ noted $\tau \preceq \sigma$, or equivalently $\sigma \succeq \tau$ iff $\exists \sigma' \in \Sigma^* : \tau; \sigma' = \sigma$. We write $\tau \prec \sigma$ (resp. $\sigma \succ \tau$) for $\tau \preceq \sigma \wedge \tau \neq \sigma$ (resp. $\sigma \succ \tau \wedge \tau \neq \sigma$). Finally, let $\tau, \sigma \in \Sigma^\infty$, τ is said to be a suffix of σ iff $\exists \sigma' \in \Sigma^* : \sigma = \sigma'; \tau$.

Following, [4], if σ has already been quantified, we freely write $\forall \tau \preceq \sigma$ (resp. $\exists \tau \preceq \sigma$) as an abbreviation to $\forall \tau \in \Sigma^* : \tau \preceq \sigma$ (resp. $\exists \tau \in \Sigma^* : \tau \preceq \sigma$). Likewise, if τ has already been quantified, $\forall \sigma \in \Sigma^\infty : \sigma \succeq \tau$ (resp. $\exists \sigma \in \Sigma^\infty : \sigma \succeq \tau$) can be abbreviated as $\forall \sigma \succeq \tau$ (resp. $\exists \sigma \succeq \tau$).

We denote by $pref(\sigma)$ (resp. $suf(\sigma)$) the set of all prefixes (resp. suffixes) of σ. Formally, $pref(\sigma) = \{\tau \in \Sigma^* \mid \tau \preceq \sigma\}$ and $suf(\sigma) = \{\tau \in \Sigma^\infty \mid \exists \tau' \in \Sigma^* : \sigma = \tau'; \tau\}$. Let $A \subseteq \Sigma^\infty$ be a subset of sequences. Abusing the notation, we let $pref(A)$ (resp. $suf(A)$) stand for $\bigcup_{\sigma \in A} pref(\sigma)$ (resp. $\bigcup_{\sigma \in A} suf(\sigma)$). The ith action in a sequence σ is given as σ_{i-1}, σ_0 denotes the first action of sequence σ, $\sigma[i, j]$ denotes the sequence occurring between actions σ_i and σ_j, and $\sigma[i..]$ denotes the remainder of the sequence, starting from action σ_i. The length of a sequence $\tau \in \Sigma^*$ is noted as $|\tau|$.

Let k be an integer. We write $\Sigma_k = \{\sigma \in \Sigma^* \mid |\sigma| = k\}$ to denote the set of sequences from Σ^* of length k, and $\Sigma_{\leq k} = \{\sigma \in \Sigma^* \mid |\sigma| \leq k\}$ to denote the set of sequences of length less or equal to k. Likewise, $pref_k(\sigma) suf_k(\sigma)$ denote the sets of prefixes and suffixes of lengths k respectively and the set of k-length factors of σ is given as $fact_k(\sigma) = \{\sigma' \in \Sigma_k \mid \exists \sigma'' \in \Sigma^*. \exists \sigma''' \in \Sigma^\infty : \sigma = \sigma''; \sigma'; \sigma'''\}$.

Finally, a security policy[3] $P \subseteq \wp\{\Sigma^\infty\}$ is a set of sets of allowed executions. A policy P is a property iff it can be characterized as a set of sequences for which there exists a decidable predicate $\hat{P}$ over the executions of Σ^∞ : $\hat{P}(\sigma)$ iff σ is in the policy [3]. In other words, a property is a policy for which the membership of any sequence can be determined by examining only the sequence itself. Such a sequence is said to be valid or to respect the property. Since all policies enforceable by monitors are properties, P and $\hat{P}$ are used interchangeably. Additionally, since the properties of interest represent subsets of Σ^∞, we follow the common usage in the literature and freely use $\hat{P}$ to refer to these sets. The distinction between a set of valid executions and the predicate over individual sequences that indicates if a given sequence is in this set only becomes relevant in Section 4.1, where we discuss the work of Schneider et al. [5].

An example of security policies that are not properties are information flow properties, which limit the way that the execution of a certain trace may influence another. More generally, policies that forbid two identical traces from occurring, or require that a certain execution is possible if another one is also allowed, are not security properties since it is impossible while examining a single execution to decide whether or not this execution respects the security policy.

[3] By security policy, we refer to the intended behavior of a software according to its formal specification. This goes well beyond security concerns and includes dependability, reliability and availability concerns as well as application-specific functionality restrictions.

A number of classes of properties have been defined in the literature and are of special interest in the study of monitoring. First are *safety* properties [6], which proscribe the occurrence of a certain "bad thing" during the execution. Let Σ be a set of actions and $\hat{P}$ be a property. $\hat{P}$ is a *safety* property iff

$$\forall \sigma \in \Sigma^\infty : \neg\hat{P}(\sigma) \Rightarrow \exists \sigma' \preceq \sigma : \forall \tau \succeq \sigma' : \neg\hat{P}(\tau). \qquad \text{(safety)}$$

Informally, this states that any sequence does not respect the security property if there exists a prefix of that sequence from which any possible extension does not respect the security policy. This implies that a violation of a safety property is irremediable: once a violation occurs, nothing can be done to correct the situation.

Alternatively, a *liveness* property [7] is a property prescribing that a certain "good thing" must occur in any valid execution. Formally, for an action set Σ and a property $\hat{P}$, $\hat{P}$ is a liveness property iff

$$\forall \sigma \in \Sigma^* : \exists \tau \in \Sigma^\infty : \tau \succeq \sigma \wedge \hat{P}(\tau). \qquad \text{(liveness)}$$

Informally, the definition states that a property is a liveness property if any finite sequence can be extended into a valid sequence. Finally, it is often useful to restrict our analysis to properties for which the empty sequence ϵ is valid. Such properties are said to be *reasonable* [4]. Formally,

$$\forall \hat{P} \subseteq \Sigma^\infty : \hat{P}(\epsilon) \Leftrightarrow \hat{P} \quad \text{is reasonable.} \qquad \text{(reasonable)}$$

An interesting result in the study of security properties is the ability to represent properties as automata. An example of this representation is the Büchi [8] automaton. A Büchi automaton is a deterministic or non-deterministic finite state automaton that accepts infinite length sequences.

Definition 1 (*From [8]*). A non-deterministic Büchi automaton is a tuple $\langle \Sigma, Q, Q_0, \delta, F \rangle$ such that

- Σ is a finite or countably infinite set of symbols;
- Q is a finite or countably infinite set of states;
- $Q_0 \subseteq Q$ is a subset of initial states;
- $\delta : Q \times \Sigma \to 2^Q$ is a (possibly partial) transition relation;
- $F \subseteq Q$ is an acceptance set. An infinite sequence ρ of symbols from Σ is valid iff at least one state present in the acceptance condition occurs infinitely often in this sequence. Such states are said to be *valid* or *accepting*. Formally, a path ρ is valid iff $inf(\rho) \cap F \neq \emptyset$ where $inf(\rho)$ is the set of states that are visited infinitely often in ρ.

Finally, we need to provide a definition of what it means to "enforce" a security property $\hat{P}$. A number of possible definitions have been suggested, yet all share that they revolve around imposing the following two principles given in [3]:

1. Soundness: All observable behaviors of the target program respect the desired property, i.e. every output sequence is present in the set of executions defined by $\hat{P}$.
2. Transparency: The semantics of valid executions is preserved, i.e. any execution of the unmonitored program that respects the security property must still be present in the monitored program. On this point, we can distinguish between precise enforcement, which does not allow any transformation of the input sequence, and equivalent enforcement, where a valid sequence can be transformed into another equivalent sequence, with respect to some equivalence relation.

3. First stage: delineating the set of enforceable properties

3.1. Security automaton modeling of monitors

The fundamental question addressed in this survey is: "which security properties are enforceable by monitors?" Schneider [3] was the first author to investigate this question. He attempted to delineate the set of security policies that are enforceable by a monitor that merely observes the execution of a target program, with no knowledge of its possible future behavior and with no ability to affect it, except by aborting the execution. He designates EM (for Execution Monitoring) the subset of security policies that could be enforced by these monitors. Although he admits that monitors could be made more powerful if these limitations were relaxed, the class he identifies as EM does serve as a lower bound to the set of security policies enforceable by monitors.

The set of policies in EM is bound by three limits inherent to runtime monitoring. Firstly, such an enforcement mechanism can only accept or reject an execution by considering it alone, without comparing it to prior runs of the same program. As discussed earlier, this is formally expressed as:

$$\exists \hat{P} : \forall \sigma \in \Sigma : \sigma \in P \Leftrightarrow \hat{P}(\sigma) \quad (1)$$

where $\hat{P}$ is a predicate over executions. This implies that only security policies which are also properties are enforceable by the mechanism under consideration.

Secondly, since at a given time the monitor does not have access to information about the possible future behavior that the program may or may not exhibit, it can never allow an invalid partial execution with the confidence that it will necessarily be corrected on all possible execution paths. The monitor can thus enforce only properties for which a violation of the property is irremediable. This is formalized as follows:

$$\forall \tau \in \Sigma^* : \neg\widehat{P}(\tau) \Rightarrow (\forall \sigma \in \Sigma^\infty : \neg\widehat{P}(\tau; \sigma)). \quad (2)$$

Thirdly, before the execution of each action by the target program, the monitor must decide either to accept it or otherwise abort the execution. It follows that any rejected execution is rejected after a finite amount of time has elapsed. This requirement is formally stated as:

$$\forall \sigma \in \Sigma^\infty : \neg\widehat{P}(\sigma) \Rightarrow (\exists i \in \mathbb{N} : \neg\widehat{P}(\sigma[0..i])). \quad (3)$$

According to the classification of security policies presented in the previous section, a policy that satisfies the requirements of Eqs. (1)–(3) is a safety property. Yet saying that monitors exactly enforce safety properties would be false on two counts. First, the analysis presented above only applies to a subset of monitors, whose behavior is especially constrained. Schneider notes that the range of enforceable security properties could be extended if the definition of a monitor was relaxed, for example by giving the monitor access to information about a program's possible behavior or the ability to alter the target program's behavior in some way. Furthermore, he also points out that other constraints, such as computability constraints, which are unavoidable, would restrict further the behavior of monitors. The set of safety properties can then best be seen as an upper bound to the properties enforceable by the simplest, most constrained monitors.

Finally, Schneider observes that safety properties can be modeled by a specific subclass of Büchi automaton, termed the *security automaton*. As discussed above, sequences of symbols model executions of the target program, and valid executions are recognized if they reach a valid state infinitely often. When restricted to safety properties, Schneider shows that a simpler definition can be used: where every state in Q is accepting and the execution is rejected as soon as an attempt is made to take an undefined transition. This alternative form of Büchi automaton was first suggested in [9], and can recognize exactly the set of safety properties. These automata have been widely used in monitoring on account of the close connection between safety properties and monitorable properties, which we outlined above.

Definition 2 (*From [3]*). A security automaton is a deterministic[4] automaton $\langle Q, q_0, \delta \rangle$ where

- Σ is a finite or countably infinite set of symbols;
- Q is a finite or countably infinite set of states;
- $q_0 \subseteq Q$ is the initial start state;
- $\delta : Q \times \Sigma \rightarrow Q$ is a (possibly partial) transition function. Rather than using an acceptance set, the execution is aborted if it attempts a transition not present in δ.

3.2. Extending the range of enforceable properties using the edit automata

Schneider's research should not be misunderstood as indicating that only safety properties can ever be enforced by monitors. Indeed, his paper [3] only discussed a specific type of monitor; one which was limited in its capacity to react to a security policy violation, and operated with no knowledge of its target's possible behavior. Schneider's study also suggested that the set of properties enforceable by monitors could be extended under certain conditions. Building on this insight, Ligatti et al. [10,11] modify Schneider's definition along three axes, namely:

1. According to the means put at the disposal of the monitor to react to a possible security policy violation. In this regard, they distinguish between monitors that can
   - abort the execution of the target program;
   - suppress a disallowed action and continue the execution;
   - insert some action or actions into the control flow;
   - insert or suppress actions in the control flow, combining the power of the previous two models.
2. According to the information made available to the monitor about the possible executions of the program. In this case, the authors distinguish between the uniform context, in which a monitor operates with no knowledge about its target's possible behavior, and the nonuniform context, in which the monitor has knowledge that certain behaviors cannot be exhibited by the target program.

[4] While the original definition in [10] allowed the security automaton to be non-deterministic, subsequent work focused on deterministic automata.

Ligatti et al. limit their analysis to finite sequences. Let $\mathcal{S}$ stand for the set of sequences which the monitor considers as possible executions of the target program. A monitor operates in a uniform context if $\mathcal{S} = \Sigma^*$, and in the nonuniform context if $\mathcal{S} \subseteq \Sigma^*$.

3. According to how much latitude the monitor is given to transform its target's execution. The authors distinguish between *precise enforcement*, which imposes that every action performed by the target program in a valid execution be preserved, and *effective enforcement* which allows a valid execution to be transformed into a semantically equivalent execution. This naturally necessitates the use of an equivalence relation $\cong$ on executions.

To model the various possible enforcement paradigms, Ligatti et al. introduce four new classes of automata. Unlike the security automata proposed by Schneider, which is designed to *recognize* whether or not an input sequence is valid, the automata proposed by Ligatti et al. are designed to *modify* the input sequence and output a new one that respects the security policy. The target program's execution is the monitor's input, which is not visible to outside observers. The automata's output represents the "corrected" behavior of the target program after the monitor has transformed it.

The execution of an automaton is specified using single step judgments of the form $(q, \sigma) \xrightarrow{\tau} (q', \sigma')$ where q is the current state, σ is the sequence the target program wishes to execute, τ is the sequence the monitor outputs on this step, q' denotes the successor state, and σ' is the input sequence after this step has been taken. Each such judgment captures a single step of the execution of the monitor. These single-step judgments are generalized to multi-step judgments of the form $(q, \sigma) \stackrel{\tau}{\Longrightarrow} (q', \sigma')$ using the two following rules:

Definition 3 (*From [11]*). A multi-step judgment $(q, \sigma) \xrightarrow{\tau} (q', \sigma')$ is constructed from single-step judgments as follows:

1. $(q, \sigma) \stackrel{\epsilon}{\Longrightarrow} (q, \sigma)$.
2. $(q, \sigma) \xrightarrow{\tau} (q'', \sigma'') \wedge (q'', \sigma'') \xrightarrow{\tau'} (q', \sigma') \Rightarrow (q, \sigma) \stackrel{\tau;\tau'}{\Longrightarrow} (q', \sigma')$.

The different enforcement paradigms studied in this research(namely: truncation, suppression, insertion and edition) are modeled using different transition relations δ. The simplest model is the truncation automaton, which simulates the behavior of Schneider's security automaton, since this automaton can only accept each input action or abort the execution. It is thus defined by a transition function δ in which every transition is restricted to those two options.[5]

Definition 4 (*From [11]*). A truncation automaton $\mathcal{A}$ is a tuple $\langle Q, \Sigma, q_0, \delta \rangle$ where

- Q is a finite or countably infinite set of states;
- Σ is a finite or countably infinite set of atomic actions;
- q_0 is the initial state, and
- $\delta : (Q \times \Sigma) \to Q$ is a (possibly partial) deterministic transition function. The automata outputs the action it has received as input if δ is defined for this action at the current state; otherwise it aborts. This behavior is specified by the following judgments:

$$(q, \sigma) \stackrel{a}{\Longrightarrow} (q', \sigma') \quad \text{if } \sigma = a; \sigma' \text{ and } \delta(a, q) = q'$$
$$(q, \sigma) \stackrel{\epsilon}{\Longrightarrow} (q', \epsilon) \quad \text{otherwise}$$

The suppression automaton possesses an added capacity to suppress actions into the output stream. This is reflected in its transition function.

Definition 5 (*From [11]*). A suppression automaton $\mathcal{A} = \langle Q, \Sigma, q_0, \delta, \omega \rangle$ where Q, Σ, q_0 and δ are defined as in 4, and $\omega : Q \times \Sigma \to \{+, -\}$ is a deterministic function with the same domain as δ that indicates whether the input action is to be output (+) or suppressed (−). This behavior is specified by the following judgments:

$$(q, \sigma) \stackrel{a}{\Longrightarrow} (q', \sigma') \quad \text{if } \sigma = a\sigma', \delta(a, q) = q' \text{ and } \omega(a, q) = +$$
$$(q, \sigma) \stackrel{\epsilon}{\Longrightarrow} (q', \sigma') \quad \text{if } \sigma = a\sigma', \delta(a, q) = q' \text{ and } \omega(a, q) = -$$
$$(q, \sigma) \stackrel{\epsilon}{\Longrightarrow} (q', \epsilon) \quad \text{otherwise}$$

The insertion automata models a monitor that can insert actions into the control flow. Its transition function thus indicates if a given input action must be output alone or alongside a finite sequence of other actions, or whether the execution must be terminated at that point.

Definition 6 (*From [11]*). An insertion automaton $\mathcal{A} = \langle Q, \Sigma, q_0, \delta, \gamma \rangle$ where Q, Σ, q_0 and δ are defined as in 4, and $\gamma : Q \times \sigma \to Q \times \Sigma^*$ is a deterministic function with a disjoint domain from that of δ. This function indicates the finite sequence which must be output for a given input sequence.

$$(q, \sigma) \stackrel{a}{\Longrightarrow} (q', \sigma') \quad \text{if } \sigma = a\sigma', \delta(a, q) = q'$$
$$(q, \sigma) \stackrel{\tau}{\Longrightarrow} (q', \sigma') \quad \text{if } \sigma = a\sigma', \text{ and } \gamma(a, q) = (\tau; q')$$
$$(q, \sigma) \stackrel{\epsilon}{\Longrightarrow} (q', \epsilon) \quad \text{otherwise}$$

Finally, the most powerful model is the edit automaton, which combines together the expressive power of the two previous automata.

Definition 7 (*From [12]*). An edit automaton $\mathcal{E}$ is a tuple $\langle Q, \Sigma, q_0, \delta \rangle$ where

- Q is a finite or countably infinite set of states;
- Σ is a finite or countably infinite set of atomic actions;
- q_0 is the initial state, and
- $\delta : (Q \times \Sigma) \to (Q \times \Sigma^\infty)$ is a (possibly partial) deterministic transition function, which indicates the output of the automaton when a given action is received as input in a given state.

Let $\mathcal{A}$ be any of the automata defined above, and let $\mathcal{A}(\sigma)$ be the output of $\mathcal{A}$ when its input is σ.

[5] In what follows, the definition of the truncation automaton, insertion, and suppression automata are lifted from [11], while that of the edit automaton is from [12].

The question of how to construct a monitor from a Büchi automaton representing a security property has been addressed several times in the literature. A constructive proof that this transformation is possible for all properties over finite sequences is given in [10]. Alternative algorithms are given in [13,14].

By using a formal model of monitors, we can define more formally the notion of enforcement. Recall that this revolves around two criteria: correctness, which imposes that the output of the monitor always be valid, and transparency, which requires that the semantics of valid executions be preserved. While the first criterion is straightforward, the second can be stated in a number of different ways. This leads to alternative notions of enforcement, which in turn induce different sets of enforceable properties for the same monitor. Two notions of enforcement are particularly interesting in this regard: *precise enforcement* and effective$_{\cong}$ *enforcement*.

A monitor precisely enforces a property if, at any step of the execution, it accepts the input action, provided that it is part of a valid sequence.

Definition 8 (*From [11]*)**.** Let Σ be a set of atomic actions and $\mathcal{S} \subseteq \Sigma^{\infty}$ be a subset of sequences. An automaton $\mathcal{A} = \langle Q, \Sigma, q_0, \delta \rangle$ precisely enforces a Property $\hat{P}$ iff $\forall \sigma \in \mathcal{S}$

1. $(\sigma, q_0) \stackrel{\sigma'}{\Longrightarrow}_{\mathcal{A}} (\epsilon, q')$;
2. $\hat{P}(\sigma')$; and
3. $\hat{P}(\sigma) \Rightarrow \forall i \in \mathbb{N} : i \leq |\sigma| : \exists q'' \in Q : (\sigma, q_0) \stackrel{\sigma'[0..i]}{\Longrightarrow}_{\mathcal{A}} (\sigma[i+1..], q'')$.

This definition is very restrictive and goes far beyond imposing syntactic equality between valid input and output. For a monitor to precisely enforce a property, it is necessary that every action present in a valid sequence be output in lockstep with the input. This condition can be relaxed to allow the monitor to transform some valid sequences as long as the output is equivalent to the input, w.r.t. some equivalence relation $\cong$.

Definition 9 (*From [11]*)**.** Let Σ be a set of atomic actions and $\mathcal{S} \subseteq \Sigma^{\infty}$ be a subset of sequences. An automaton $\mathcal{A}$effectively$_{\cong}$ enforces a Property $\hat{P}$ iff $\forall \sigma \in \mathcal{S}$

1. $(\sigma, q_0) \stackrel{\sigma'}{\Longrightarrow}_{\mathcal{A}} (\epsilon, q')$;
2. $\hat{P}(\sigma')$; and
3. $\mathcal{A}(\sigma) \cong \sigma'$.

The equivalence relation $\cong$ can capture any semantic property of sequences. Ligatti et al. only impose that it respects an *indistinguishability* requirement.

$$\forall \hat{P} : \forall \sigma, \sigma' \in \Sigma^{\infty} : \sigma \cong \sigma' \Rightarrow (\hat{P}(\sigma) \Leftrightarrow \hat{P}(\sigma')). \quad \text{(indistinguishability)}$$

In [15], Chabot shows that syntactic equality is the only equivalence relation that respects this definition.

By contrasting various combinations of the different possibilities enumerated above, Ligatti et al. provide a rich taxonomy of security policies, in association with appropriate enforcement models. This also illustrates how each of the factors discussed above can influence the set of policies that can be enforced by a monitor.

The simplest model presented in this regard is that of a truncation automaton that precisely enforces a property on a uniform system. This case is identical to that of the security automaton discussed above, which has been shown to enforce exactly the set of safety properties. However, in the nonuniform case, some liveness properties can be enforced. This happens when static analysis or code instrumentation assures the monitor that any partial sequence that violates the property will eventually be corrected. The monitor can thus allow the invalid sequence to proceed with the confidence that any violation will eventually be corrected. The set of policies effectively applicable by truncation automata is also greater than what this mechanism can precisely enforce. Similar to the benefit of operating in the nonuniform context, a gain occurs when any valid sequences with invalid prefixes can be thought of as equivalent to one of its prefixes, which in turn exhibits only valid prefixes.

The set of properties enforceable by the truncation automaton in these three cases can serve as a baseline to examine the enforcement power of the other automata. In a uniform system, the set of policies precisely enforceable by the suppression, insertion and edit automata is the same as that of the truncation automaton. It is perhaps surprising that the ability to suppress or insert an action gives the monitor no added power. This is a result of the narrow definition of precise enforcement: it requires that executions which respect the security policy be accepted without modification, and that each action be output by the automaton in lockstep with the program. Since it is impossible to both modify an execution and respect this requirement, the added abilities of the more elaborate insertion, suppression and insertion automata cannot extend the range of precisely enforceable policies in a uniform system.

For precise enforcement in the nonuniform context, however, Ligatti et al. found an incremental increase in the set of enforceable properties with the suppression automaton being strictly more powerful than the truncation automaton, and the insertion automaton being strictly more powerful than the suppression automaton. Interestingly, they found that on a nonuniform system, the insertion automaton can precisely enforce any property that can be precisely enforced by the suppression automaton. This is because the insertion automaton can mimic the behavior of the suppression automaton in the following way: if the suppression automaton suppresses an action a, it can do so because it knows, from static analysis, that the prefix of the execution that has already elapsed could be extended possibly by several suffixes. The insertion automaton has only to affix one of these suffixes and terminate the execution to ensure the respect of the security policy. There are also properties that are precisely enforced by the insertion automaton but not by the suppression automaton. These cases occur when the insertion automaton uses its ability to insert actions into the control flow to correct an otherwise invalid sequence. Likewise, on nonuniform systems, the edit automaton enforces exactly the same set of properties as the insertion automaton. This result is not surprising since the edit automaton is a combination of the insertion and suppression automaton, and the insertion automaton can enforce in this context a superset of the properties applicable by the suppression automaton.

These results do not hold for effective enforcement, where the enforcement power of the insertion and suppression

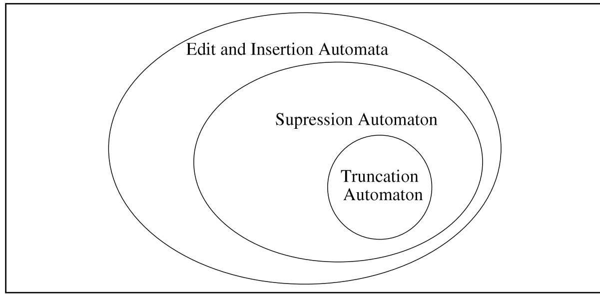

Fig. 1 – Precise application on nonuniform systems from [10].

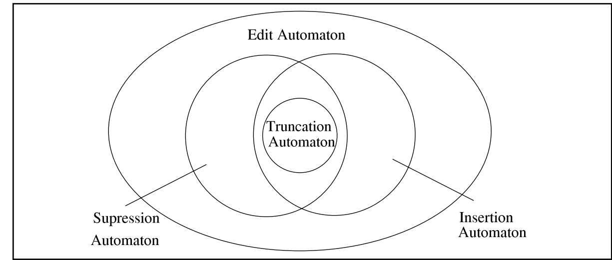

Fig. 2 – Effective$_\cong$ application from [10].

automata are orthogonal. This observation is a bit surprising in light of the previous discussion and follows from the fact that in certain cases, no actions can be inserted in the sequence and at the same time preserve the semantics of the sequence intact, whereas this result can be achieved if one or more actions are suppressed. Finally, the set of properties effectively enforced by the edit automaton is a superset of the set of properties enforceable by any other enforcement paradigm we have previously considered. The power of the edit automaton derives from its ability to suppress an input sequence as long as it is potentially invalid and to reinsert it if it becomes valid. Since this analysis is limited to finite sequences, and the empty sequence is always assumed to be valid, the edit automaton can effectively enforce any property.

The taxonomy presented above is summarized in Figs. 1 and 2 from [10]. Two observations emerge particularly from these figures. The first is that the range of security policies enforceable by most paradigms of monitoring is extended in the nonuniform context, as opposed to the uniform context, with all other factors remaining equal. Secondly, a monitor capable of more varied responses when faced with a violation of the security property may be able to enforce a wider range of security properties, but only if it is given sufficient latitude to alter a valid input sequence by its equivalence relation. Alternatively, if the equivalence relation is too strict and the monitor is unable to transform valid sequences, the added power of the monitor is lost and the set of enforceable security properties is not extended.

3.3. *The edit automaton and infinite sequences*

The analysis from [10] has focused exclusively security policies stated over finite sequences. Yet the behavior of many systems is possibly nonterminating. In [4,16], Ligatti et al. extend the above analysis to a more general framework containing both finite and infinite sequences. They focus on effective$_\cong$ enforcement by the edit automaton since this is the more powerful framework. Effective enforcement depends on an equivalence relation between executions. Since these relations tend to be highly system-specific, Ligatti et al. focus on syntactic equality (=). It can be argued that any other equivalence relation is more inclusive; the set of policies effectively$_=$ enforceable is thus a lower bound on the set of policies that can be effectively enforced by the edit automaton.

Intuitively, the effective$_=$ enforcement by the edit automaton works in the following way: as long as the actions input by the target program comply with the security policy, they are immediately output. If, however, the automaton detects that the input actions are part of a potentially malicious sequence, the edit automaton may suppress them (outputting nothing), and then store the input sequence until it is certain that they do indeed respect the security policy, at which point the automaton will re-insert them. On the other hand, if the automaton determines that the input sequence is irrevocably invalid, it terminates the execution having output only a valid prefix of the input sequence. The monitor is in fact *simulating* the execution of the program until it is certain that the behavior it has so far witnessed is correct. For the time being, it is assumed that the monitor has an unlimited ability to do this for all program actions. In [14], we see that a monitor behaving in this manner always outputs the longest valid prefix of its input if the latter is invalid.

The authors define the set of infinite renewal properties (*Renewal*) in order to characterize the set of properties enforceable by an edit automaton in the following way. A property is a member of this set if every infinite valid sequence has infinitely many valid prefixes, while every invalid infinite sequence has only finitely many such prefixes. Formally, for an action set Σ and a property $\hat{P}$, $\hat{P}$ is a *Renewal* property iff it meets the following two equivalent conditions:

$$\forall \sigma \in \Sigma^\omega : \hat{P}(\sigma) \Leftrightarrow \{\sigma' \preceq \sigma \mid \hat{P}(\sigma')\} \text{ is an infinite set} \qquad (renewal_1)$$

$$\forall \sigma \in \Sigma^\omega : \hat{P}(\sigma) \Leftrightarrow (\forall \sigma' \preceq \sigma : \exists \tau \preceq \sigma : \sigma' \preceq \tau \wedge \hat{P}(\tau)). \qquad (renewal_2)$$

Note that the definition of Renewal imposes no restrictions on the finite sequences in $\hat{P}$. This is consistent with the result in [10]: on systems containing only finite sequences, every property is enforceable by the edit automaton. For infinite sequences, the set of Renewal properties includes all safety properties, some liveness properties and some properties which are neither safety nor liveness. For example, a property that states that a given action must eventually occur in any non-empty execution fits this definition since any valid infinite length execution has infinitely many valid prefixes (prefixes in which that action occurs) while any invalid infinite length execution has only a finite number of valid prefixes (none). The set of infinite renewal properties, contrasted to that of safety and liveness, is given in Fig. 3.

In this regard, Falcone et al. [14] provided the insight that the class of renewal properties is equivalent to the union of four of the six classes of the safety-progress classification of properties [17], namely safety, guarantee, obligation, and response; the property classes of reactivity and persistence contain properties which are not effectively$_=$ enforceable

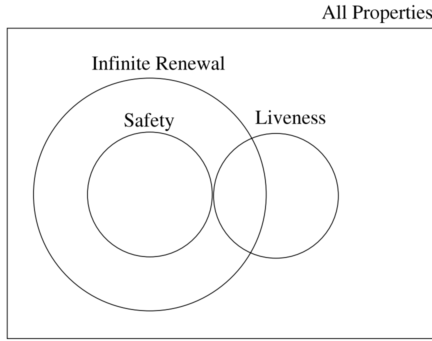

Fig. 3 – The set of infinite renewal properties.
***Source:* From [16].**

by the edit automaton. This classification is an alternative to the safety-liveness dichotomy, in which properties are arranged in a hierarchy of six classes. The authors further give algorithms to construct a monitor from a property for each of the 4 classes for which this is possible.

The set of reasonable infinite renewal properties is a lower bound to the set of properties that can be effectively$_=$ enforced by the edit automaton. This mechanism also enforces properties that are not in Renewal. This happens when, at a certain point during the execution, the automaton determines that the current prefix of the execution it has been fed can only be extended in a single sequence that respects the security policy. In this case, it can output that sequence and terminate the execution immediately. Ligatti et al. however, regard this case as a marginal corner case.

It is also important to emphasize that effective$_=$ enforcement is only a lower bound on the set of properties that can be effectively enforced by the edit automaton: an enforcement mechanism that effectively$_=$ enforces a property does not rely upon the power of the edit automaton to transform an execution to make it conform with the security requirement. Rather than reject an illegal execution, the automaton could either delete some troublesome actions from the control flow and output a modified, valid sequence, or it could add an action that must inevitably happen if the execution is to be valid, thus extending the set of enforceable properties. Furthermore, the set renewal is only a lower bound to the set of properties enforceable in a nonuniform context. As shown in the previous section, the set of properties enforceable by this mechanism is extended when the monitor operates in the nonuniform context, which allows non-renewal properties to be enforced.

While the automata-based model of monitors presented throughout this survey is the most widely used in the literature, other models have been proposed. In [18], Zhu et al. suggest modeling monitors by way of a *stream automata* while another alternative model, the *Mandatory results Automata* is suggested by Ligatti et al. in [19]. Both of these models differ from the edit automaton in that they distinguish between the action set of the target and that of the system with which it interacts. These models make it easier to study the interaction between the target program, the monitor and the system. In this study, while we continue to focus on the truncation and edit automaton, we believe the results we present can easily be applied to the more refined automata models introduced in these papers. In [20], Jun models monitors as Mealy Machines [21] and studies their expressive power.

3.4. *Comparing mechanisms' enforcement power*

Drawing on the work of Ligatti et al., Chabot [15], in his study of the truncation automaton, suggests a notation to describe the set of properties enforceable by a given automaton type. Extending Chabot's notation to cover every model proposed in [10], we let $\{\mathcal{T}, \mathcal{D}, \mathcal{I}, \mathcal{E}, \mathcal{A}\}$ range over the possible enforcement mechanisms, namely truncation ($\mathcal{T}$), suppression ($\mathcal{D}$) (for deletion), insertion ($\mathcal{I}$), edition ($\mathcal{E}$), or any enforcement monitor ($\mathcal{A}$). Let $\mathcal{S} \subseteq \Sigma^\infty$ stand for a subset of possible execution sequences, and let $\cong$ be an equivalence relation between the sequences of Σ^∞; we write $\mathcal{A}^{\mathcal{S}}_{\cong}$-effectively enforceable to denote the set of properties which are effectively$_{\cong}$ enforceable by a monitor of class $\mathcal{A}$ when the set of possible sequences is $\mathcal{S}$ and $\cong$ is the equivalence relation limiting the monitor's ability to transform its input. Likewise, we write $\mathcal{A}^{\mathcal{S}}$-precisely enforceable for the set of properties which are precisely enforceable by a monitor of class $\mathcal{A}$ when the set of possible sequences is $\mathcal{S}$. We omit the superscripted set of possible input sequences when this set is Σ^∞.

We can illustrate the above notation quite easily. For example, $\mathcal{E}^{\Sigma^\infty}_{=}$-effectively enforceable represents the set of properties which are effectively enforceable by an edit automaton in the uniform context, when the equivalence relation is syntactic equality, while $\mathcal{D}^{\mathcal{S}}$-precisely enforceable is the set of properties which are precisely enforceable by the suppression automaton when the set of possible input sequences is $\mathcal{S}$.

In light of the above, we are now in a position to translate the main results from [10,11] in an original formalism that exhibits the advantages of concision and precision. Theorem 1 below indicates that in the case of uniform precise enforcement, only safety properties are enforceable, regardless of the capabilities of the monitor.

Theorem 1 (*From* [11]). *$\mathcal{T}^{\Sigma^*}$-precisely enforceable $= \mathcal{I}^{\Sigma^*}$-precisely enforceable $= \mathcal{D}^{\Sigma^*}$-precisely enforceable $= \mathcal{E}^{\Sigma^*}$-precisely enforceable $=$ Safety.*

Theorem 2 indicates that for all classes of monitors, the set of precisely enforceable properties is extended in the uniform context. We should stress that this result does not imply that $\forall S \subset \Sigma^* : \mathcal{M}^{\mathcal{S}}$-precisely enforceable $\subset \mathcal{M}^{\Sigma^*}$-precisely enforceable. Indeed, a counterexample proving that this is not the case is given by Chabot et al. in [22].

Theorem 2 (*From* [11]). *$\forall \mathcal{M} \in \{\mathcal{T}, \mathcal{D}, \mathcal{I}, \mathcal{E}\} : \exists S \subset \Sigma^* : \mathcal{M}^{\Sigma^*}$-precisely enforceable $\subset \mathcal{M}^{\Sigma^S}$-precisely enforceable.*

The relative power of each class of automata monitor, when operating in the context of nonuniform precise enforcement is given in Theorem 3, while that of monitors operating in a context of uniform effective$_=$ enforcement is given in Theorems 4 and 5.

Theorem 3 (*From* [11]). *$\forall S \subset \Sigma^*$: Safety $\subset \mathcal{T}^{\mathcal{S}}$-precisely enforceable $\subset \mathcal{D}^{\mathcal{S}}$-precisely enforceable $\subset \mathcal{D}^{\mathcal{S}}$-precisely enforceable $= \mathcal{E}^{\mathcal{S}}$-precisely enforceable.*

Theorem 4 (*From* [11]). *(Safety* $\subset$ $\mathcal{T}^{\Sigma^*}$*-effectively enforceable* $\subset$ $\mathcal{I}^{\Sigma^*}$*-effectively enforceable.)* $\wedge$ *(*$\mathcal{T}^{\Sigma^*}$*-effectively enforceable* $\subset$ $\mathcal{D}^{\Sigma^*}$*-effectively enforceable).*

Theorem 5 (*From* [11]). *(*$\mathcal{I}^{\Sigma^*}$*-effectively enforceable* $\subset$ $\mathcal{E}^{\Sigma^*}$*-effectively enforceable)* $\wedge$ *(*$\mathcal{D}^{\Sigma^*}$*-effectively enforceable* $\subset$ $\mathcal{E}^{\Sigma^*}$*-effectively enforceable).*

Finally, Theorem 6 shows that an edit automaton can effectively$_=$ enforce any property stated over finite sequences.

Theorem 6 (*From* [11]). $\forall \hat{P} \subseteq \Sigma^* : \hat{P} \in \mathcal{E}^{\Sigma^*}$*-effectively enforceable.*

In sum, the outcome of initial studies in this first stage of investigations into the enforcement power of monitors was that the most constrained monitors are capable of enforcing exactly the set of safety properties. Subsequent studies showed how the set of monitorable properties can be extended thought different paths until nearly all properties are monitorable. Yet, However, it should be noted that all studies which we have classified in this first stage ignore the presence of memory and computational constraints that can limit a monitor's power. These aspects have been addressed by the studies we have grouped in the second stage of development to which we now turn.

4. Second stage: monitoring with memory and computability constraints

Having examined the enforcement capacities of various classes of monitors, we now turn our attention to the question of how these capabilities are affected by computability and memory constraints. Indeed, both Schneider and Ligatti et al. have stated that such constraints might make a property unenforceable by a given security mechanism even if it lies inside the set of properties enforceable by this mechanism. For example, a safety property may not be enforceable by a truncation automaton because the monitor is unable to detect the violation of the security property when it occurs. As such, the results presented in the previous sections should be regarded as upper-bounds to the set of properties enforceable by each mechanism. In this section, we review studies that examine exactly how such constraints affect the set of enforceable properties.

4.1. *Imposing computability constraints*

Initial work in this direction was done by Kim et al. [23] who observe that for a security automaton to be able to enforce a property, it needs to be able to identify any invalid sequence upon the inspection of a finite prefix of this sequence. It follows that monitors enforce only those properties for which such an inspection is possible. Formally:

Theorem 7. *A property* $\hat{P}$ *is enforceable by a security automaton iff* $\hat{P}$ *is a safety property and the set* $\Sigma^* \setminus pref(\hat{P})$ *is recursively enumerable.*

In other words, a property is enforceable by a security automaton if it is a safety property, and the set of executions that do not respect the property is recursively enumerable. As observed in [24], this result can also be given as stating that the set of enforceable properties is the class of co-recursively enumerable properties (coRE) (Fig. 4).

Proceeding along a similar line of inquiry, Hamlen et al. [5] compare the set of properties enforceable by three enforcement mechanisms and link them to known classes from computational complexity theory. In order to model the (possibly infinite) execution of the target program, they introduce Program Machines (PM), deterministic state machines, akin to Turing Machines (TM) that can accept an infinite input and exhibit an output sequence rather than accepting or rejecting this input. In keeping with previous sections of this study, we let σ, τ range over both input and output sequences and use $M(\sigma)$ to refer to the output of PM M when it is given σ as input. We further let X_M be the set of all possible output sequences of M, and $pref(X_M)$ be that of finite prefixes of sequences in X_M.

The results Hamlen et al. presented in their study contrast the enforcement power of monitors with those of two other enforcement mechanisms, namely static analysis and code rewriting. Consequently, although our study focuses on runtime monitors specifically, it behooves us to introduce their results with respect to these two enforcement mechanisms as well.

Hamlen et al. first consider the case of static analysis. A property $\hat{P}$ can be statically enforceable if and only if there exists a TM $M_{\hat{P}}$ that takes as input a PM M and either accepts or rejects in finite time, according to whether or not $\hat{P}$ holds on M. This definition is both intuitive and broad enough to encompass any enforcement mechanism based on static analysis. This definition also matches exactly that of the class of recursively decidable properties (known as the class Π_0 in the arithmetic hierarchy) for which there exists a total computable procedure that decides them. Since every such policy is also in coRE, statically enforceable properties form a subset of monitorable properties. This result is based on the premise that any static analysis could be performed by the monitor at the onset of a program's execution.

Next, Hamlen et al. examine the class $RW_{\approx}$ of properties enforceable by program rewriters: mechanisms that, in infinite time, modify an untrusted program prior to its execution to ensure its compliance with a security property. This category includes the more expressive monitors such as the edit automaton. Like Ligatti et al. in [10], they observe that the power of such an enforcement mechanism can only be examined in the context of a specific equivalence relation, which constrains the rewriter's ability to transform a program. The authors suggest defining an equivalence relation $\approx$ between PMs, which in turn is defined in term of an equivalence relation $\cong$ over the possible executions of the PM.

Let M_1 and M_2 be two PMs; M_1 is equivalent to M_2, noted $M_1 \approx M_2$ if and only if:

$$\forall \sigma \in \Sigma^\infty : M_1(\sigma) \cong M_2(\sigma). \quad (4)$$

The following two constraints on $\cong$ are imposed:

$$M_1(\sigma) \cong M_2(\sigma) \text{ is recursively decidable if } M_1(\sigma) \text{ and } M_2(\sigma) \text{ are finite} \quad (5)$$

$$\sigma \cong \sigma' \Rightarrow (\forall i \in \mathbb{N} : \exists j \in \mathbb{N} : \sigma[0..i] \cong \sigma'[0..j]). \quad (6)$$

The first requirement ensures that a procedure be able to determine whether or not two executions are equivalent. The latter imposes that equivalent executions have equivalent prefixes. Given a certain equivalence relation $\approx$, property $\hat{P}$ can be enforceable by rewriting if there exists a total computable rewriting function, $R \subseteq PM \times PM$ that can transform any PM into a new valid (with respect to $\hat{P}$) PM while preserving the semantics of every valid PM. Formally:

$$\hat{P}(R(M)) \quad (7)$$

$$\hat{P}(M) \Rightarrow M \approx R(M). \quad (8)$$

After careful examination, we see that this definition is equivalent to that of effective$_\cong$ enforcement given in [10]. Since no restriction is imposed on the PM's output in those cases where the execution is invalid, every satisfiable statically enforceable property is trivially RW$_\approx$ enforceable. We can determine statically which programs respect this property, and the rewriting relation R has only to return the program unchanged if it respects the property or some other arbitrarily chosen valid execution otherwise. Only the unsatisfiable property, which rejects all executions, is statically enforceable but cannot be enforced by program rewriters in this manner since there is no valid execution which can be substituted for the invalid ones. Policies that are RW$_\approx$ enforceable also include some but not all properties in coRE and some properties not in this class.

Turning their attention back to the properties enforceable by monitoring, Hamlen et al. argue that the class coRE actually represents an upper-bound to the set of properties enforceable by this mechanism. Note first that when a monitor detects a violation of the security property, it must react to prevent this violation. In their model, this intervention is specified by a sequence of actions (possibly the end of program token) that the monitor appends to the current input sequence. Let I be the set of all possible interventions by the monitor. The property $\hat{P}_I$, which forbids all such interventions, cannot be enforced by a monitor. Yet this property is in coRE if I is a computable set. For example, the non termination property $\hat{P}_{\text{end}}$, which demands that the target program's execution does not terminate, is not enforceable by a truncation automaton since the only remedial course at the disposal of such a monitor is explicitly forbidden.

Even more remarkably, the authors observe that the same property can become enforceable or not by monitors depending on how the predicate $\hat{P}$ that characterizes this property is stated. For example, consider the property P_{RU} which states that a resource cannot be used (by the action e_{use}) after it has been released (by the action e_{rel}). This property can be stated as

$$\hat{P}_{RU1}(\sigma) = (\forall i \in \mathbb{N} : i \geq 1 : e_{\text{rel}} \notin \sigma[0..i] \vee e_{\text{use}} \notin \sigma[i+1..]). \quad (9)$$

This predicate states that no e_{use} can occur after an e_{rel} action. This property can be enforced by a suppression automaton by suppressing the occurrence of e_{use}, or by a truncation automaton, by aborting the execution.

Alternatively, this same property can be given as

$$\hat{P}_{RU2}(\sigma) = (e_{\text{rel}} \notin \sigma \vee (\forall \sigma' \in \Sigma^{\infty} : \sigma; \sigma' \in \Sigma^{\infty} : e_{\text{use}} \notin \sigma')). \quad (10)$$

This predicate states that any execution containing an e_{rel} action cannot be extended to include a e_{use} action. Both are the same property in the sense that both define the same execution set. While this is a safety property in coRE, the question of whether or not a finite execution will be extended to respect $\hat{P}_{RU2}$ is undecidable. This is because while both predicates rule out the same executions, $\hat{P}_{RU2}$ actually defines a violation of the property in any execution by an event that occurs earlier than $\hat{P}_{RU1}$. A truncation mechanism that relies upon the definition of $\hat{P}_{RU1}$ is thus unable to prevent some violations of the policy from occurring.

A better characterization of the set of properties enforceable by monitors actually is the intersection of the class coRE and the set RW$_\approx$. Properties in this intersection exhibit a particular behavior termed *benevolence*. A property is benevolent if there exists a decision procedure $M_{\hat{P}}$ that rejects any invalid prefix of an invalid execution, but accepts any valid prefix of a valid execution, for all possible PM M. This allows the monitor to reject valid prefixes if it determines that they will be extended to invalid executions. Observe that the monitor is not required to accept the valid prefixes of invalid executions. Formally, a property $\hat{P}$ is benevolent if there exists a decision procedure $M_{\hat{P}}$ such that for all PM M:

$$\neg(\forall \sigma \in X_M : \hat{P}(\sigma)) \Rightarrow (\forall \sigma \in \mathit{pref}(X_M) : (\neg \hat{P}(\sigma) \Rightarrow M_{\hat{P}}(\sigma) \text{ rejects})) \quad (11)$$

$$(\forall \sigma \in X_M : \hat{P}(\sigma)) \Rightarrow (\forall \sigma \in \mathit{pref}(X_M) : (M_{\hat{P}}(\sigma) \text{ accepts})). \quad (12)$$

$\hat{P}_{RU1}$ is an example of a benevolent property.

Properties that do not lie in the intersection of coRE and RW do not have the benevolence property, and thus cannot be meaningfully enforced by monitors. In the absence of a decision procedure $M_{\hat{P}}$ that fits the description above, the monitor is required to either reject some valid executions, or to only reject a violation of the security property after it has occurred. This result squares with our intuition that monitors can be inlined in the program they aim to protect, so that any property enforceable by monitor can also be enforced by program rewriting.

A summary of the findings of Schneider et al. is given in Fig. 5. As discussed earlier, any statically enforceable property is seen as being also enforceable by monitoring and program rewriting, under the assumption that a monitor can perform a static analysis of its target before its execution begins. Note however that this presupposes that both the static and the dynamic analyzer have access to the same information about the program's behavior. Static analyzers often have access to source code unavailable to monitors, which could allow them to enforce a greater range of properties. The set of properties enforceable by monitors is defined as the intersection of coRE properties and RW$_\approx$ properties. Properties within the intersection of these classes exhibit a useful behavior, termed benevolence, which allows the monitor to reject all invalid behaviors before they occur. This result indicates that the class of monitorable properties is somewhat smaller than

previous studies have shown. This inconstancy is explained by the fact that previous studies considered that a monitor can enforce some properties even though the monitor only discovered a violation of the property only after it had occurred and thus outputs an invalid sequence. Finally, the set of properties enforceable by program rewriting is shown to be a superset of that of properties enforceable by monitoring, owing to the rewriters greater ability to transform the target program. This is consistent with the conclusion of Ligatti et al. [10] who argue that the edit automaton, which can alter a program's semantics in the manner of a rewriter, is strictly more powerful than a truncation or insertion automaton, which lacks this ability.

4.2. *Monitoring and memory constraints*

Among the assumptions made previously when studying the enforcement power of monitors is the assumption that they are unconstrained by any memory limitations. In practice, however, it is reasonable to assume that monitors have only a finite amount of computational resources at their disposal. Several authors have thus examined the set of properties enforceable by a monitor whose finite memory allows it to keep only a partial record of its ongoing target execution. Despite this limitation, memory-constrained monitors where shown to be capable of enforcing several practical real-life properties. Researchers have proposed three distinct ways to represent this limitation: monitors with a memory bounded to some finite value k, monitors with a finite but unbounded memory and monitors that record only the unordered set of actions that have occurred in the program so far.

4.2.1. *Shallow history automata*

The study of memory-constrained automata and their use in monitoring was pioneered by Fong [25]. He introduces the Shallow History automaton (SHA), which only records the shallow history (i.e. the unordered set of security relevant events performed by the target program).

Definition 10 (*From [25]*). A Shallow History automaton is a tuple $\langle \Sigma, F(\Sigma), H_0, \delta \rangle$ where

- Σ is a finite or countably infinite set of events;
- $F(\Sigma)$ is the set of all possible shallow histories;
- $H_0 \in F(\Sigma)$ is an initial access history, usually $\emptyset$;
- $\delta : F(\Sigma) \times \Sigma \rightarrow F(\Sigma)$ is a transition function, which is defined as $\delta(H, a) = H \cup \{a\}$ if δ is defined for $\langle H, a \rangle$.

The set of properties enforceable by SHA, known as EM_{SHA}, is a strict subset of the set EM of properties enforceable by Schneider's security automaton. Yet, several interesting real-life properties can be enforced under this paradigm. Fong gives the following 4 examples: The Chinese Wall Policy [26], the Low-Water-Mark policy [27], One-out-of-k authorization [28] and the Assured Pipelines policy [29,30].

Fong further suggests that other meaningful subclasses of EM could be formed by abstraction, i.e. by merging several states of a SA into a single abstract state, making them indistinguishable from the perspective of policy enforcement. Let A be a set of abstract states and $\alpha : \Sigma^* \rightarrow A$ be an abstraction function. An automaton whose states are built from applying the abstraction function α to the states of a security automaton SA is noted SA_α. A property is enforceable by a SA_α if it can be enforced using only information left behind after the abstraction process. We write EM_α for the set of properties enforceable by SA_α. The SHA, as well as the set of properties it can enforce, can be seen as an instance of this abstraction process generated by a particular abstraction function. Every abstraction α induces an equivalence relation $\equiv_\alpha$ such that $\forall \sigma, \sigma' \in \Sigma^* : \sigma \equiv_\alpha \sigma' \Leftrightarrow \alpha(\sigma) = \alpha(\sigma')$.

In Section 5, we will show how subsequent research drew upon Fong's use of equivalence relations in order to provide a model of enforcement that better describes the behavior of real-life monitors.

The use of equivalence relations provides an intuitive comparison point between the subsets of EM. Let $EM_\equiv$ be the set of properties enforceable by $\equiv$. Fong shows that more properties are enforceable by a monitor bound by an equivalence relation $\equiv_1$ than one bound by $\equiv_2$ if $\equiv_1$ is more differentiating than $\equiv_2$.

Theorem 8 (*From [25]*). *Let $\equiv_1, \equiv_2$ be equivalence relations. $\equiv_1 \subseteq \equiv_2 \Rightarrow EM_{\equiv_1} \subseteq EM_{\equiv_2}$ Furthermore, if the former inclusion is strict, so is the latter.*

This comparison also argues for a lattice-based classification of the subclasses of *EM* defined by abstraction. Let the join operator be the intersection of two equivalence relations ($\equiv_1 \sqcup \equiv_2 \;=\; \equiv_1 \cap \equiv_2$) and meet the transitive closure of the union of two equivalence relations ($\equiv_1 \sqcap \equiv_2 \;=\; (\equiv_1 \cup \equiv_2)^+$). These two operators define a lattice over the set of all equivalence relations over a given set of actions Σ^*. The top element of this lattice ($\equiv_\top$) corresponds to the class EM, and the bottom element ($\equiv_\bot$) corresponds to the class of memoryless properties enforceable by an automaton with a single state. The classes of properties enforceable by a SHA and by any other subclass of EM are ordered on a poset induced by the above lattice classification.

The following are corollaries of Theorem 8.

Corollary 9 (*From [25]*). $EM_\bot \subset EM_{SHA} \subset EM_\top$

Furthermore, Fong shows that the set of properties enforceable by a Shallow history automaton is a strict subset of the set of properties enforceable by an unbounded truncation automaton. Formally:

Corollary 10 (*From [25]*). $EM_{SHA} \subset \mathcal{T}^{\Sigma^*}$*-precisely enforceable.*

4.2.2. *Bounded history automata*

Following Fong's line of inquiry, and linking it to Ligatti's research on the edit automata, Talhi et al. [12,31,32] devised the Bounded History Automaton (BHA), which has only a limited space with which to store the program's execution history and showed how this new automata class can still enforce a relevant set of properties despite this limitation. They further showed how bounded-memory monitors can be modeled both by Bounded Security Automata (BSA) or by Bounded Edit Automata (BEA), analogous to the security and edit automata introduced by Schneider [3] and Ligatti[10] respectively.

Bounded history automaton (BHA) is a class of automata used to model security properties enforceable by monitors

equipped with only a bounded memory with which to store the past history of the target execution. Each state of the automaton thus represents a finite-size abstraction of the input sequence that has occurred so far. Talhi et al. define two subclasses of BHA: The Bounded Security Automaton, which is analogous to a truncation automaton and can only allow an action or abort the execution, and the Bounded Edit Automaton, analogously to an edit automaton, which can also suppress and insert actions into the input stream.

Definition 11 (*From [12]*). A Bounded Security Automaton of bound k (noted k-BSA) is a tuple $\langle \Sigma, Q = \Sigma_{\leq k}, q_0, \delta \rangle$ where

- Σ is a finite or countably infinite set of input actions.
- Q is a finite or countably infinite set of states, each of which represent a bounded history of size smaller or equal to k.
- k is the maximal size of the history.
- $q_0 \in Q$ is the initial state. This is usually the empty history ϵ.
- $\delta : (Q \times \Sigma) \to Q$ is the (possibly partial) transition function. If for a given automaton $A, \delta(h, a) = h'$, this signifies that the bounded history h' is an abstraction of $h; a$, and that h' contains all the information necessary for the enforcement of the property captured by A.

Analogous to an edit automaton, a BHA can be enriched with the ability to insert or suppress actions in the programs they monitor. The history stored by a bounded edit automata consists of two sequences: the first has been output by the automaton, while the second is suppressed until a valid prefix is recognized.

Definition 12 (*From [12]*). A Bounded edit automaton of bound k (noted k-BEA), is a tuple of the form $\langle \Sigma, Q = (\Sigma_{\leq k}, \Sigma_{\leq k})_{\leq k}, q_0, \delta \rangle$ where:

- Σ is a finite or countably infinite set of input actions.
- Q is a finite or countably infinite set of states, each of which is a pair $\langle \sigma_{Acc}, \sigma_{Sup} \rangle$ such that $\sigma_{Acc}, \sigma_{Sup} \in \Sigma_{\leq k}$.
- k is the maximal size of the history.
- $q_0 \in Q$ is the initial state. This is usually the empty history. $\langle \epsilon, \epsilon \rangle$.
- $\delta : (Q \times \Sigma) \to Q$ is the (possibly partial) transition function. As with the BSA, if $\delta(h, a) = h'$ then h' is an abstraction of $h; a$ containing all the relevant information necessary for the enforcement of the property accepted by the BEA.

Talhi et al. prove a number of theorems related to the expressivity of this enforcement mechanism. Let EM_{kSA} be the set of properties enforceable by BSA, and EM_{kEA} be the set of properties enforceable by BEA. EM_{kSA} is naturally a subset of safety properties enforceable by unbounded SAs, and EM_{kEA} a subset of reasonable infinite renewal properties, enforceable by unbounded EAs. The set of properties enforceable by bounded automata increase monotonously as a larger memory is made available to the monitor. Formally:

Theorem 11 (*From [12]*). *For any two integers k and k' such that $k < k'$, we have $EM_{kSA} \subset EM_{k'SA}$ and $EM_{kEA} \subset EM_{k'EA}$.*

BSA are equally expressive as the SA_α defined by Fong. For any BSA, there exists a SA_α, and conversely, for any SA_α, there exists a k-bound BEA with some maximal memory size k that enforces the same property. Furthermore, any SHA can be translated into a k-bound BEA where k is the cardinality of the set Σ of input actions, provided that Σ is finished. These results are formalized in Theorems 12–14.

Theorem 12 (*From [12]*). *For any BSA enforcing a Property $\hat{P}$, there exists a SA_α that also enforces $\hat{P}$.*

Theorem 13 (*From [12]*). *For any SA_α enforcing a Property $\hat{P}$, there exists a k-bound BSA $\mathcal{A}$ enforcing the same property, where k is the maximal size of the histories of $\mathcal{A}$.*

Theorem 14 (*From [12]*). *Let $\mathcal{A} = \langle \Sigma, F(\Sigma), H_0, \delta \rangle$ be a SHA enforcing a Property $\hat{P}$, if Σ is a finite set and $|\Sigma| = k$; there exists a k-bound BSA that enforces the same property.*

There exists a close connection between the set of properties enforceable by BHA and a class of properties termed locally testable properties (or local properties) [33]. Locally testable properties are recognizable by a class of automata termed *scanners*, which are equipped with a finite memory and a sliding window of size k. This window is slid along the input sequence starting with the initial action; only the actions in the window are visible to the scanner at any given time. Properties are enforceable by scanners if they can be recognized using only the prefixes and suffixes of length less than k and the factors (subsequences) of length k. In [34], it is observed that such properties are particularly well suited to be enforced by monitoring because a monitor enforcing such a property needs only to keep a record of the last k computational cycles. Furthermore, if a monitor enforces several such properties, this record can be shared so the memory overhead of the monitor is independent of the number of locally testable properties being enforced.

Let P, S, X and F be 4 sets such that $P, S \subseteq \Sigma_{k-1}, X \subseteq \Sigma_{\leq k-1}$ and $F \subseteq \Sigma_k$.

Definition 13 (*From [12]*). A property L is k-locally testable if and only if it respects the following two rules:

- $\forall \sigma \in \Sigma^* : \sigma \in L \setminus \{\epsilon\} : (\sigma \in X) \vee ((pref_{k-1}(\sigma) \cap P \neq \emptyset) \wedge (suf_{k-1}(\sigma) \cap S \neq \emptyset) \wedge (fact_k(\sigma) \subseteq F))$.
- $\forall \sigma \in \Sigma^\omega : \sigma \in L \setminus \{\epsilon\} : \forall \sigma' \in pref(\sigma) : \exists \sigma'' \in pref(\sigma) : \sigma' \in pref(\sigma'') \wedge \sigma'' \in L$.

Informally, a property $\hat{P}$ is k-locally testable if any finite non-empty sequence $\in \hat{P}$ is either of size less than k or if it has both a prefix and a suffix in the sets P and S respectively, and that all their factors of size k are in the set F. Taken together, these requirements mean that a scanner examining this property through a sliding window of size k will always be able to recognize that the sequence under consideration is in the $\hat{P}$.

A property L is locally testable if it is k-locally testable for some integer k. There exists several algorithms for deciding if a language is locally testable and finding the minimal k value for which it is k-locally testable [35–38]. These algorithms are polynomial in time with respect to the size of the language's alphabet and that of the automaton used to represent the language.

The set of locally testable properties intersects with that of prefix testable, suffix testable, and prefix–suffix testable properties, and it contains a subset termed strongly locally testable properties [39]. Prefix testable properties are properties that can be recognized by examining only prefixes of the input sequences; conversely suffix testable properties are recognizable by examining only suffixes of the input sequences. Prefix–suffix testable properties are recognizable by examining both prefixes and suffixes of the input sequences. Each of these classes contains a subset of locally testable properties the k-prefix, k-suffix and k-prefix–suffix testable properties, which can be recognized by examining only a bounded prefix, suffix, or a prefix and a suffix of size k. Finally, strongly locally testable properties are a subset of locally testable properties which are recognizable by inspecting factors of a bounded size.

Talhi et al. express the connection between locally testable properties and properties enforceable by BSA in a set of four theorems presented below as Theorems 15–18. Interpreting their results, we see that since the BSA is a subclass of the SA, properties enforceable by the BSA will necessarily be a subset of those enforceable by SAs, and thus will be prefix-closed. Prefix-closed k-prefix testable properties are enforceable by k-bounded BSA, so are k-strongly enforceable properties and locally testable properties in general if they are prefix-closed. Conversely, suffix testable and prefix–suffix testable properties are generally not enforceable by BSA.

Theorem 15 (*From* [12]). *Any prefix-closed k-prefix testable property $\hat{P}$ is enforceable by some k-BSA.*

Theorem 16 (*From* [12]). *Any k-strongly locally testable property $\hat{P}$ is enforceable by some k-BSA.*

Theorem 17 (*From* [12]). *Any prefix-closed k-locally testable property $\hat{P}$ is enforceable by some k-BSA.*

Theorem 18 (*From* [12]). *Any suffix testable and prefix–suffix testable properties are not enforceable by BSA.*

Performing a similar analysis on the enforcement power of the BEA, Theorems 19–22 show that the more powerful BEA enforces strictly more properties. A k-bounded BEA can enforce any k-prefix testable property, any k-strongly enforceable property and any k-locally testable properties. Suffix testable and prefix–suffix testable properties are generally not enforceable by BEA.

Theorem 19 (*From* [12]). *Any prefix-closed k-prefix testable property $\hat{P}$ is enforceable by some k-BEA.*

Theorem 20 (*From* [12]). *Any k-strongly locally testable property $\hat{P}$ is enforceable by some k-BEA.*

Theorem 21 (*From* [12]). *Any k locally testable property $\hat{P}$ is enforceable by some k-BEA.*

Theorem 22 (*From* [12]). *Any suffix testable and prefix–suffix testable properties are not enforceable by BSA.*

The above eight theorems allow us to link the properties enforceable by memory bounded automata with classes of formal languages. This aids us in the task, when presented with a real-life property, of deciding which class of monitor is apt to enforce it. Talhi et al.'s contribution in this context is three-fold. First, they devise a new abstract model of monitors, the Bounded History Automaton, and explore its enforcement power. Second, they devise a new taxonomy of enforceable properties based on the amount of memory needed for the enforcement. Thirdly, they show a connection between locally testable properties and those enforceable by BHA. Their research allows us to better characterize the properties enforceable by monitors constrained by memory limitations.

4.2.3. *Finite automaton*

A final inquiry into the set properties enforceable by an edit automaton with memory constraints was done by Beauquier et al. In [40], they examine the set of properties enforceable by an edit automaton with a finite, but unbounded, set of states. They focus on effective enforcement and on uniform systems containing both finite and infinite sequences with = as the equivalence relation and they define a new class of properties, termed memory bounded properties, which coincides with the set of properties that are effectively$_=$ enforceable by a finite edit automaton.

Before presenting the result of this study, we introduce the specific notation that was used by the authors in this regard. Let Σ be a set of atomic actions and $\hat{P} \subseteq \Sigma^\infty$ be a property. We write $\hat{P}_{\text{fin}}$ for $\hat{P} \cap \Sigma^*$ and $\hat{P}_{\text{inf}}$ for $\hat{P} \cap \Sigma^\omega$. $\overrightarrow{\hat{P}_{\text{fin}}}$ designates the set of infinite sequences which have infinitely many prefixes in $\hat{P}_{\text{fin}}$.

Definition 14 (*From* [40]). A property $\hat{P}$ is memory bounded if $\hat{P}$ is of the form $\hat{P}_{\text{fin}} \cup \overrightarrow{\hat{P}_{\text{fin}}} \bigcup_{j \in J} R_j; \beta_j$ where

- $\epsilon \in \hat{P}$;
- J is finite;
- $\hat{P}_{\text{fin}}$ is regular;
- $\forall j \in J : R_j$ is regular;
- $\forall j \in J : \beta_j$ is an ultimately periodic sequence in Σ^ω;
- $\forall j \in J : R_j \cap pref(\hat{P}_{\text{fin}}) = \emptyset$;
- $\forall i, j \in J : i \neq j = R_j \cap pref(R_i) = \emptyset$
- there exists a constant K such that for every $u \prec v$ s.t. $u \notin pref(\hat{P}_{\text{fin}})$ and $v \in R_j$ we have $|v| - |u| \preceq K$

We can now state the main result from [40] using our notation introduced in Section 3.4.

Theorem 23 (*From* [40]). *Let $\mathcal{F}_{=}^{\Sigma^\infty}$-effectively enforceable stand for the set of properties that are effectively enforceable by a finite edit automaton in a uniform context with = as the equivalence relation. A Property $\hat{P}$ is in $\mathcal{F}_{=}^{\Sigma^\infty}$-effectively enforceable iff $\hat{P}$ is memory bounded.*

Contrasting this result with that of Ligatti et al. [4] and of Tahli et al. [12] we can say that the set of $\mathcal{F}_{=}^{\Sigma^\infty}$-effectively enforceable properties is both a subset to that of Renewal properties enforceable by an edit automaton unconstrained by memory limitation and a superset to the set of properties enforceable by the more constrained BEA.

Taken together, the results of studies in the second stage of investigations into the enforcement power of monitors show that despite memory limitations, monitors are still able to enforce a relevant subset of security properties. Since real-life monitors will necessarily exhibit some bound on the memory use of computing power, these results confirm the applicability of monitors in practice.

5. Third stage: alternative definitions of enforcement

In the final stage of inquiry into the enforcement power of monitors, researchers have revisited the notion of *enforcement* by a monitor. Indeed recent research has argued that the original definition of effective enforcement [10] is inadequate because it does not sufficiently constrain the behavior of the monitor when it is faced with a possible violation of the security policy. These works propose alternative definitions and examine the set of properties enforceable when they are used.

5.1. *Defining subclasses to effective enforcement*

In [41,42], Bielova et al. revisit an example of a property that is $\mathcal{E}_{=}^{\Sigma^\infty}$-effectively enforceable given by Ligatti et al. in [11]. This is the "market policy" given as follows: Let take(n) and pay(n) be two atomic actions that represent the acquisition and corresponding payment of *n* apples respectively. The policy states that any apple that is taken must be paid for, either before or after the acquisition takes place (take(n);pay(n) or pay(n);take(n)). In [11], produce an automaton that effectively$_=$ enforces this property. As discussed above, the automaton enforces the property by suppressing the execution as long as its input is invalid and outputting the current prefix when a valid sequence is reached.

Neither Ligatti et al. nor Bielova et al. give a formal definition of the predicate $\hat{P}$ defining the market policy, but Bielova et al. observe that any reasonable definition would impose that any consecutive pair of actions of the form take(n);pay(n) or pay(n);take(n) present in the input also be present in the output, since this represents the purchase of *n* apples by a customer. But consider the behavior of the system when presented with the sequence pay(1); browse; pay(2);take(2). The automaton suggested by Ligatti et al. outputs nothing, since this automaton is limited to outputting prefixes of its input sequence, which is made irremediably invalid by the presence of a pay(1) action lacking a corresponding take(1) action. Indeed, this meets the definition of effective$_=$ enforcement since ϵ is the longest valid prefix of this sequence. However, the valid transaction pay(2);take(2) is not present in the output.

What Bielova et al. realized is that the definition of effective enforcement is inadequate in that it makes no demands on the behavior of the monitor when it is presented with an invalid input sequence other than that it must output a valid sequence. An invalid sequence may be substituted for any valid sequence. In practice, a monitor would most likely be bounded to preserve some aspect of the original input or limited in its ability to insert new behaviors not present in the input sequence. Bielova et al. concisely state the problem in a subsequent paper: [13]: "*What distinguishes an enforcement mechanism is not what happens when traces are good, because nothing should happen! The interesting part is how precisely bad traces are converted into good ones. To this extent soundness only says they should be made good. The practical systems, being practical, will actually take care of correcting the bad traces. But this part is simply not reflected in the current theories.*"

Bielova et al. suggest several constraints that can be imposed on the behavior of monitors for both valid and invalid inputs. This allows them to define subclasses of the edit automaton and compare their enforcement power.

The first of these subclasses is the Delayed Automaton, thus named because it only delays the appearance of input actions until a valid prefix has been built up. Such an automaton never outputs an action not present in its input.

Definition 15 (*From [41]*[6]). A Delayed Automaton $\mathcal{A}$ is an edit automaton as defined in Definition 7, with the added restriction that

$$\forall \sigma \in \Sigma^* : \mathcal{A}(\sigma) \preceq \sigma.$$

The automaton defined in [10] is also limited in that at every step, the monitor either outputs all suppressed actions or suppresses the current action and outputs nothing. In other words, this monitor never outputs only some of the actions it has previously suppressed but not yet output, and it never outputs a previously suppressed action if the current action is being suppressed. Bielova et al. refer to an automaton operating in this manner as an All-Or-Nothing automaton.

Definition 16 (*From [41]*). An *All-Or-Nothing Automaton* $\mathcal{A}$ is an edit automaton as defined in Definition 7 with the added restrictions that

$$\forall \sigma \in \Sigma^* : \mathcal{A}(\sigma) \preceq \sigma \quad \text{and}$$
$$\forall \sigma \in \Sigma^* : : \forall a : \mathcal{A}(\sigma; a) = \sigma; a \vee \mathcal{A}(\sigma; a) = \mathcal{A}(\sigma).$$

A third subclass of the edit automaton can be defined by imposing yet another restriction, namely that the automaton always output the longest valid prefix of its input sequence. Such an automaton is termed a *Ligatti's automaton for property* $\hat{P}$ where $\hat{P}$ is the property the monitor enforces. The automaton suggested in [10] to enforce the market policy is of this type.

Definition 17 (*From [41]*). Let $\hat{P}$ be a property. A *Ligatti's automaton for property* $\hat{P}$ $\mathcal{A}$ is an edit automaton as defined in Definition 7 with the added restrictions that

$$\forall \sigma \in \Sigma^* : \mathcal{A}(\sigma) \preceq \sigma; \forall \sigma \in \Sigma^* : \forall a \in \Sigma : \mathcal{A}(\sigma; a) = \sigma; a \vee \mathcal{A}(\sigma; a) = \mathcal{A}(\sigma) \text{ and}$$
$$\forall \sigma \in \Sigma^\infty : \hat{P}(\sigma) \Rightarrow \mathcal{A}(\sigma) = \sigma.$$

[6] Because we introduced the edit automaton in Section 3.1 using the definition given by Talhi et al. in [12], while Bielova et al. rely upon the equivalent original definition from [11], the definitions of the automata occurring in this section have been adapted.

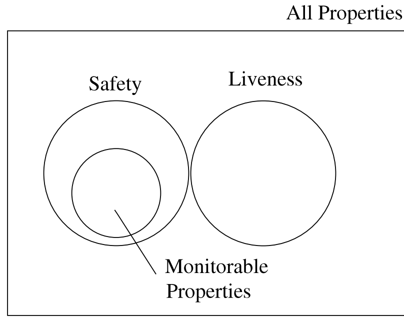

Fig. 4 – Properties enforceable by monitors from [23].

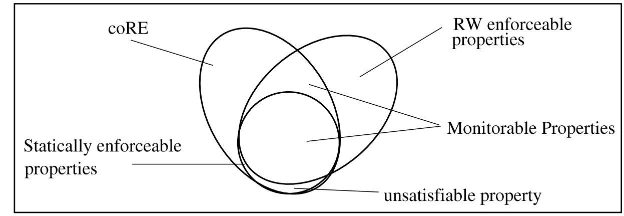

Fig. 5 – Properties enforceable by various mechanisms from [5].

A final subclass of the edit automaton can be defined by limiting the monitor to output a valid prefix of the input when the latter is valid but leaving the monitor unconstrained otherwise. Such an automaton is called a *Delayed automaton for property* $\hat{P}$.

Definition 18 (*From [41]*). Let $\hat{P}$ be a property. A *Delayed automaton for property* $\hat{P}$ $\mathcal{A}$ is an edit automation as defined in Definition 7 with the added restriction that $\forall \sigma \in \Sigma^* : \mathcal{A}(\sigma) \preceq \sigma \vee \hat{P}(\mathcal{A}(\sigma))$.

Since Bielova et al. found the definition of effective$_{\cong}$ to be inadequate to enforce the market policy, they suggest an alternate notion of enforcement called *delayed precise enforcement*. This enforcement paradigm captures a requirement that the monitor must always output a valid sequence and that if the input sequence is valid, the output must be syntactically equal to it; furthermore, the output must always be a prefix of the input.

Definition 19 (*From [41]*). Let Σ be a set of atomic actions. An automaton $\mathcal{A} = \langle Q, \Sigma, q_0, \delta, \omega\rangle$ delayed precisely enforces a Property $\hat{P}$ iff $\forall \sigma \in \Sigma^\infty$

1. $(\sigma, q_0) \overset{\sigma'}{\Longrightarrow}_{\mathcal{A}} (\epsilon, q')$;
2. $\hat{P}(\sigma')$; and
3. $\hat{P}(\sigma) \Rightarrow \sigma = \sigma' \wedge \forall i \in \mathbb{N} : \exists j \in \mathbb{N} : j \leq i \exists q* \in Q : (\sigma, q_0) \overset{\sigma[0..j]}{\Longrightarrow}_{\mathcal{A}} (\sigma[i+1..], q*)$.

This definition can be seen as intermediate between precise enforcement and effective$_{=}$ enforcement and captures the manner in which Ligatti et al. propose to enforce security properties in [16]. Any automaton which delayed precisely enforces a Property also effectively$_{=}$ enforces this property.

Comparing the classes of automaton introduced above, we find that the class of Delayed automaton and Delayed automaton for property $\hat{P}$ intersect, but that neither is a strict subset of the other. An all-or-nothing automaton $\mathcal{A}$ is a Delayed automaton but is not necessarily a Delayed automaton for property $\hat{P}$. Likewise a delayed automaton, even if it delayed precisely enforces a property $\hat{P}$ is not necessarily a Ligatti automaton for property $\hat{P}$. However, the class of edit automaton which effectively$_{=}$ enforces a property $\hat{P}$ is a subclass of Delayed automaton for property $\hat{P}$. Furthermore, the class of edit automaton which delayed precisely enforces a Property $\hat{P}$ coincides with the intersection between the class of Delayed automaton, Delayed automaton for property $\hat{P}$ and the class of edit automaton which effectively$_{=}$ enforces the property $\hat{P}$. Finally, an all-or-nothing automaton enforces a property $\hat{P}$ iff it is also a Ligatti automaton for property $\hat{P}$.

Theorem 24 (*From [41]*). *If an automaton* $\mathcal{A}$ effectively$_{=}$ *enforces a property* $\hat{P}$, *then* $\mathcal{A}$ *is a Delayed automaton for property* $\hat{P}$ *but is not necessarily a Delayed automaton.*

Theorem 25 (*From [41]*). *An automaton* $\mathcal{A}$ *delayed precisely enforces a Property* $\hat{P}$ *iff* $\mathcal{A}$ *is a Delayed automaton,* $\mathcal{A}$ *is a Delayed automaton for property* $\hat{P}$, *and* $\mathcal{A}$ effectively$_{=}$ *enforces the property* $\hat{P}$.

Theorem 26 (*From [41]*). *An all-or-nothing automaton* $\mathcal{A}$ *precisely enforces the property* $\hat{P}$ *iff* $\mathcal{A}$ *is also a Ligatti automaton for property* $\hat{P}$.

These results are summarized in Fig. 6.

Bielova et al. then turn their attention to monitors capable of producing an output which is not a prefix of its input. They focus on a specific class of properties termed iterative properties, which model the repeated execution of transactions.

Definition 20 (*From [13]*). A property $\hat{P}$ is an iterative property iff $\forall \sigma, \sigma' \in \Sigma^* : \hat{P}(\sigma) \wedge \hat{P}(\sigma') \Rightarrow \hat{P}(\sigma, \sigma')$.

Iterative properties serve to model the desired behavior of systems that are intended to repeatedly perform a number of finite transactions, each of which can be seen as atomic. The typical example is the software of an online store or an ATM. Stack inspection is an iterative property, as is termination and access control. More generally, the class of iterative properties includes some but not all safety, liveness and renewal properties. This is illustrated in Fig. 7.

A new notion of enforcement, tailored to the application of such iterative properties, can be explained as follows. Informally, a monitor *iteratively enforces by suppression* an iterative property $\hat{P}$ iff every valid transaction is output and every invalid transaction is suppressed. This idea is defined more formally as follows:

Definition 21 (*From [13]*). Let Σ be an action set and let $\hat{P}$ be an iterative property. Automaton $\mathcal{A}$ iteratively enforces by suppression property $\hat{P}$ if

1. $\forall \sigma \in \Sigma^* : \hat{P}(\sigma) \Rightarrow \mathcal{A}(\sigma) = \sigma$.
2. $\forall \sigma \in \Sigma^* : \neg\hat{P}(\sigma) : \exists \sigma' \succeq \sigma : \hat{P}(\sigma') \Rightarrow \mathcal{A}(\sigma) = \sigma_0$ where σ_0 is the longest valid prefix of σ.
3. $\forall \sigma \in \Sigma^* : \neg\hat{P}(\sigma) \wedge \forall \sigma' \succeq \sigma : \neg\hat{P}(\sigma') : \exists \sigma_b \in \Sigma : \sigma = \sigma_0; \sigma_b\sigma_r \Rightarrow \mathcal{A}(\sigma) = \sigma_0; \mathcal{A}(\sigma_r)$ where σ_0 is the longest valid prefix of σ, and σ_b is the smallest sequence s.t. after deleting it from σ, the resulting sequence is valid.

Bielova et al. show how to construct an edit automaton which iteratively enforces a transaction property $\hat{P}$. The research of Bielova et al. also confirms a prior result from [10], namely that the enforcement power of monitors is intricately linked to the definition of enforcement they use. It follows that any restriction imposed on the monitor's ability to alter invalid sequences must be crafted carefully. If designed too narrowly, it will unduly restrict the range of enforceable properties. Conversely, designed too leniently, it could result in monitors that enforce the property, but not in a manner that is desirable or useful.

5.2. *Corrective enforcement*

The work of Bielova et al. raises a number of thought provoking points. First, they have identified a shortcoming in the notion of enforcement used in most other studies: effective$_{\cong}$ enforcement does not place any restrictions on the monitor's behavior when it is faced with an invalid sequence. In practice, we may be interested in *how* a monitor deals with such a situation. This raises several new questions, such as: Which valid sequence will be chosen as a replacement? How is this new sequence chosen? Which aspect of the original invalid sequence has to be preserved, and which has to be deleted or replaced?

Answering these questions is central to the development of practical monitoring software. Indeed, a monitor which effectively$_{\cong}$ enforces a property by always replacing any invalid sequence with the same arbitrarily chosen valid sequence (perhaps ϵ) will not be useful in many critical systems where it is essential that the execution not terminate. In practice, one would expect a monitor to perform only the minimal alterations necessary to transform an invalid sequence into a valid one. Valid behaviors present in an invalid sequence should be preserved whenever possible, and no more should be added or subtracted to the original sequence than what is necessary to assure the compliance of the execution with the security policy. Providing this kind of enforcement requires that the behavior of the monitor be constrained when it encounters an invalid sequence.

The solution advanced in [42] in the case of iterative properties is ingenious and innovative. In effect, it imposes that valid transactions present in the original sequence also be present in the output, thus constraining the monitor's behavior. Yet this solution creates a few drawbacks. More particularly, the enforcement paradigm is specific to the property being enforced (iterative enforcement and iterative properties). Enforcing a different property requires a different enforcement paradigm. A more general enforcement paradigm, that is parameterized to fit the desired property, as effective enforcement is parameterized with an equivalence relation $\cong$, would be more practical.

Two such alternative enforcement paradigms have been proposed by Khoury et al. in [43,44]. The first framework groups together related sequences into equivalence classes, and then limits the monitor so that it can only return an output sequence equivalent to the input.

Following previous work in monitoring by Fong [25], they use an abstraction function $\mathcal{F} : \Sigma^* \rightarrow \mathcal{I}$ to capture the property of the input sequence which the monitor must preserve throughout its manipulation. The set $\mathcal{I}$ can be any abstraction of the program's behavior. Such an abstraction can capture any property of relevance. This may be, for example, the presence of certain subwords or factors or any other semantic property of interest. It is this property which must be preserved by the monitor despite any modification performed to assure the respect of the security policy. To this end, the equivalence relation groups together sequences sharing the same abstraction as well as sequences for which the abstraction is similar (w.r.t. the security policy of interest).

This kind of enforcement is termed corrective$_{\cong}$ enforcement, since its focus is on producing an output sequence which meaningfully corrects the violation occurring in the input sequence. Formally:

Definition 22 (*From [43]*). Let $\mathcal{A}$ be an edit automaton, and let $\cong$ be an equivalence relation. $\mathcal{A}$correctively$_{\cong}$ enforces the property $\hat{P}$ iff $\forall \sigma \in \Sigma^{\infty}$

1. $\hat{P}(\mathcal{A}(\sigma))$.
2. $\mathcal{A}(\sigma) \cong \sigma$.

A monitor can correctively$_{\cong}$ enforce a property iff for every possible sequence there exists an equivalent valid sequence which is either finite or has infinitely many valid prefixes, and the transformation into this sequence is computable.

It is important to stress how the use of equivalence relation in corrective$_{\cong}$ enforcement differs from that in effective$_{\cong}$ enforcement developed by Ligatti et al. Indeed, if an equivalence relation meets the condition indistinguishability of indistinguishability, which Ligatti identified as the minimal condition a equivalence relation must meet to be of use in effective$_{\cong}$ enforcement, then it becomes impossible to use this equivalence relation for corrective$_{\cong}$ enforcement. If any two equivalent sequences always meet this criterion, an invalid prefix can never be made valid by replacing it with another equivalent one. It is thus impossible to "correct" an invalid prefix and output it.

Instead, an equivalence relation used for corrective$_{\cong}$ enforcement must be consistent with the abstraction rather than with the security property. Formally:

$$\mathcal{F}(\sigma) = \mathcal{F}(\sigma') \Rightarrow \hat{P}(\sigma) \Leftrightarrow \hat{P}(\sigma'). \quad (13)$$

Both iterative enforcement by suppression and effective$_{=}$ enforcement can be stated as special cases of the more general corrective$_{\cong}$ enforcement paradigm. In the case of effective$_{=}$ enforcement, the required equivalence relation is $\cong_{\preceq}$, which is defined such that $\forall \sigma, \sigma' \in \Sigma^* : \sigma \cong_{\preceq} \sigma' \Leftrightarrow pref(\sigma) \cap \hat{P} = pref(\sigma') \cap \hat{P}$ for some property $\hat{P}$. Using this relation two sequences are equivalent, w.r.t. a given property $\hat{P}$ iff they have the same set of valid prefixes.

Theorem 27 (*From [43]*). *A property $\hat{P}$ is* effectively$_{=}$ *enforceable iff it is* correctively$_{\cong_{\preceq}}$ *enforceable.*

As was the case with previous enforcement frameworks the set of property that are correctively$_{\cong}$ enforceable using a given equivalence relation, can be extended by the use of static analysis. This set can also be extended by using a coarser equivalence relation.

An equivalence relation $\cong$ over a given set Σ^* can be seen as a set of pairs (x, y) with $x, y \in \Sigma^*$. This allows equivalence relations over the same sets to be compared. Relation $\cong_1$ is a refinement of relation $\cong_2$, noted $\cong_1 < \cong_2$ if the set of pairs in $\cong_1$ is a strict subset of those in $\cong_2$.

Theorem 28. *Let* $\cong_1, \cong_2$ *be two equivalence relations and let enforceable* $_\cong$ *stand for the set of properties which are* correctively$_\cong$ *enforceable; then we have* $\cong_1 < \cong_2 \Rightarrow$ enforceable$_{\cong_1} \subset$ enforceable$_{\cong_2}$.

By limiting the monitor's ability to transform invalid sequences, corrective$_\cong$ provides a more meaningful enforcement paradigm. However, this definition also raises some difficulties. In particular, it implies that several distinct valid sequences, which are possible transformations of an invalid sequence, must be equivalent. Likewise, it requires that several invalid sequences be considered equivalent if a single valid sequence is a valid alternative to both.

Furthermore, the method of enforcement is also encoded into the equivalence relation. An automaton enforcing an iterative property by transforming invalid transactions into valid ones would require a different equivalence relation than one enforcing the same property by suppressing invalid transactions. This makes it harder to compare alternative enforcement paradigms of the same property.

In [44], Khoury et al. propose an alternative notion of enforcement termed corrective$_\sqsubseteq$ enforcement, which addresses these issues by replacing the use of equivalence relations with partial orders. As was the case with corrective$_\cong$ enforcement, an abstraction function is used to capture any aspect of the execution sequence that the monitor must preserve. The abstractions are then organized along a partial order, rather then partitioned into equivalence classes. Sequences are accordingly organized into a partial order $\sqsubseteq$. The monitor is permitted to transform an input σ into another sequence σ' iff $\sigma \sqsubseteq \sigma'$.

Definition 23 (*From [44]*). Let $\mathcal{A}$ be an edit automaton and let $\sqsubseteq$ be a partial order over the sequences of Σ^∞. $\mathcal{A}$ correctively$_\sqsubseteq$ enforces the property $\hat{P}$ iff $\forall \sigma \in \Sigma^\infty$

1. $\hat{P}(\mathcal{A}(\sigma))$
2. $\sigma \sqsubseteq \mathcal{A}(\sigma)$

The use of partial orders solves the problems mentioned above. Furthermore, the authors observe that partial orders are a more natural way to express security policies than equivalence relations. Naturally, corrective$_\cong$ enforcement can be seen as a special case of this framework. The main drawback of this enforcement paradigm is that the restrictions on the treatment of invalid sequences must be defined by the user for each property. Since these restrictions relate to the semantics of the property being enforced, they cannot be straightforwardly transferred from one property to another.

In [45,46] Bielova et al. suggest an alternative solution to this problem in which the restriction on the monitor's treatment of invalid sequences follows constraints that are syntactic in nature. They term this idea the *predictability* of enforcement. Informally, an enforcement mechanism is predictable if it transforms every invalid input sequence that is close to a valid one into a sequence close to that same valid one (using some metric and threshold to define "close"). The authors then propose the Levenshtein distance as a possible candidate metric. Formally, predictability in enforcement is defined as follows:

Definition 24 (*From [45]*). Let $\mathcal{A}$ be an automaton and let d and d' be a measure of the distance between two sequences. A enforcement mechanism is predictable within some distance ε iff for every trace $\sigma \in \hat{P}$, the following holds: $\forall \nu \in \mathbb{R} : \nu \geq \varepsilon : \exists \delta \in \mathbb{R} : \delta > 0 : \forall \sigma' \in \Sigma^* : (d(\sigma, \sigma') \leq \delta \Rightarrow d'(\mathcal{A}(\sigma), \mathcal{A}(\sigma')) \leq \nu)$.

Informally, this definition states that for every valid sequence, there exists a radius δ such that all the traces within this radius are transformed into traces that occur within a radius ε of the same sequence. The main advantage of this enforcement paradigm over the ones proposed by Khoury et al. is that the former does not require that the constraints treatment of invalid sequences be specified explicitly for each property, thus allowing for a more automatic generation of the enforcement mechanism from the property.

6. Conclusion

In this paper, we have surveyed various studies that examine the question of which security policies are enforceable by monitors. Research on this topic has followed three consecutive lines of inquiry. The first stage of research has been to delineate the set of properties monitors are capable of enforcing, and to suggest various ways this set can be extended. These include giving the monitor access to statically gathered data about the target program's possible behavior, or giving the monitor more diverse means to react to a potential violation of the security policy. Subsequent research examined constraints that limit the monitor's enforcement power, such as memory and computability constraints. Finally, a third stage of research found that the very definition of *enforcement* being used can greatly affect the set of properties that can be enforced. The analysis of these three stages confirms that monitors are capable of enforcing a wide range of security policies.

While we now have a clearer idea of which security properties monitors are capable of enforcing, several important questions remain open.

- An important avenue for future research is to study the interaction between abstraction and enforceable properties. Indeed, real-life monitors may generate a very large quantity of information, and any enforcement mechanism that relies upon the traces they generate will necessarily have to abstract these traces into a higher formalism before analyzing them. However, information lost to this abstraction process may render certain security properties unenforceable, leading to the need to balance the trade-off between abstraction and enforcement.
- Another promising avenue for future research is the problem of monitor certification stated in [47] as the problem of proving (and giving guarantees of) the soundness of a monitor. Doing so would remove the potentially complex and policy-specific monitor from the trusted code base and replace it with a simpler and policy-independent certifier.
- As discussed above, static analysis is an effective tool for extending the set of enforceable properties in most monitoring contexts. This occurs when static analysis rules out the occurrences in the system of execution paths whose presence in the system make the property of interest unenforceable. An important question that still be

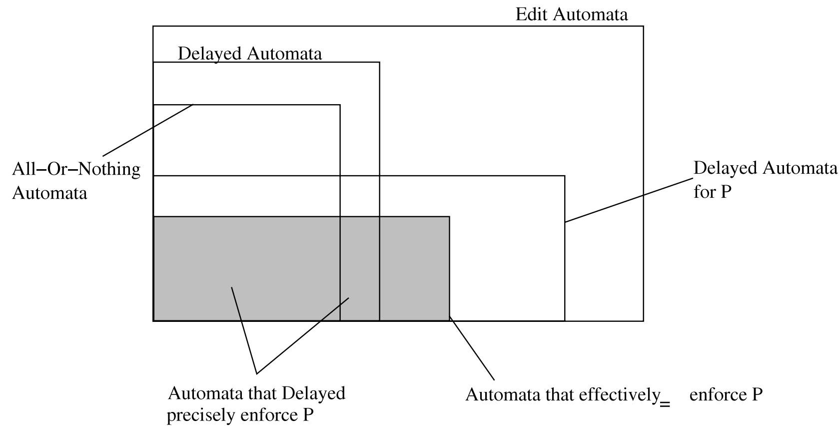

Fig. 6 – Comparing the various subclasses of edit automata.
***Source*: From [41].**

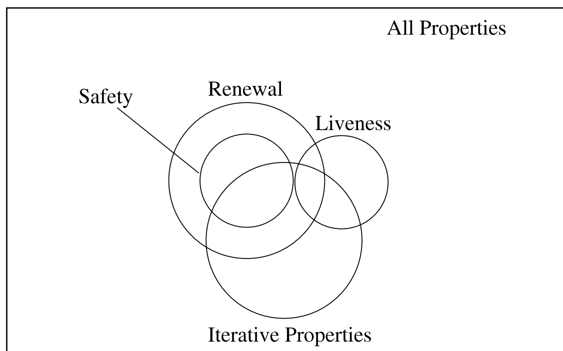

Fig. 7 – Iterative properties.
***Source*: From [13].**

answered: for a given property $\hat{P}$, what is the least property $\hat{P}$' that must be successfully verified by a target system $\mathcal{S}$ so that $\hat{P}$ becomes $\mathcal{T}_{\cong}^{\mathcal{S}}$-effectively enforceable or $\mathcal{E}_{\cong}^{\mathcal{S}}$-effectively enforceable? Answering this question would make it easier to build hybrid static–dynamic security policy enforcement mechanisms with wider ranges of enforceable security policies.

- The edit automaton, the most powerful model of monitors, relies on an unlimited capacity to suppress an ongoing execution and reinsert it later as it becomes apparent that the execution under observation is valid. However, as observed in [4], it is not always possible for the monitor to suppress an action and observe the subsequent execution of the target program, as the latter may be contingent on the result of the evaluation of the former. This reality constrains the edit automata power, but by exactly how much remains an open question. Likewise, a real-life system may also contain security-critical actions that must originate with the user and cannot be inserted by the monitor if they are absent in the original sequence. Delineating the enforcement power of the edit automata in the presence of actions that cannot be suppressed or cannot be inserted is an important topic for future research.

REFERENCES

[1] A. Bauer, M. Leucker, C. Schallhart, Monitoring of real-time properties, in: Proceedings of the 26th Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS, in: Lecture Notes in Computer Science, vol. 4337, Springer-Verlag, Berlin, Heidelberg, 2006.
[2] A. Bauer, M. Leucker, C. Schallhart, Runtime verification for LTL and TLTL, ACM Transactions on Software Engineering and Methodology (2011).
[3] F. Schneider, Enforceable security policies, Information and System Security 3 (1) (2000) 30–50.
[4] J. Ligatti, L. Bauer, D. Walker, Run-time enforcement of nonsafety policies, ACM Transactions on Information and System Security 12 (3) (2009).
[5] K.W. Hamlen, G. Morrisett, F.B. Schneider, Computability classes for enforcement mechanisms, ACM Transactions on Programming Languages and Systems 28 (2006) 175–205.
[6] L. Lamport, Proving the correctness of multiprocess programs, IEEE Transactions on Software Engineering 3 (2) (1977) 125–143.
[7] B. Alpern, F. Schneider, Defining liveness, Information Processing Letters 21 (4) (1985) 181–185.
[8] S. Eilenberg, Automata, Languages and Machines, Vol. A, Academic Press, New York, 1974.
[9] B. Alpern, F. Schneider, Recognizing safety and liveness, Distributed Computing 2 (1987) 17–126.
[10] L. Bauer, J. Ligatti, D. Walker, More enforceable security policies, in: Foundations of Computer Security, Copenhagen, Denmark, 2002.
[11] J. Ligatti, L. Bauer, D. Walker, Edit automata: enforcement mechanisms for run-time security policies, International Journal of Information Security (2005).
[12] C. Talhi, N. Tawbi, M. Debbabi, Execution monitoring enforcement under memory-limitations constraints, Information and Computation 206 (1) (2008) 158–184.
[13] N. Bielova, F. Massacci, A. Micheletti, Towards practical enforcement theories, in: Identity and Privacy in the Internet Age, 14th Nordic Conference on Secure IT Systems, NordSec 2009, Oslo, Norway, 14–16 October 2009, Proceedings, 2009, pp. 239–254.
[14] Y. Falcone, J.-C. Fernandez, L. Mounier, Synthesizing enforcement monitors wrt. the safety-progress classification of properties, in: Information Systems Security, 4th International

Conference, ICISS 2008, Hyderabad, India, December 16–20, 2008, Proceedings, in: Lecture Notes in Computer Science, vol. 5352, 2008, pp. 41–55.
[15] H. Chabot, Sécurisation de code basée sur la combinaison de la vérification statique et dynamique, Master's Thesis, Laval University, 2009.
[16] J. Ligatti, L. Bauer, D. Walker, Enforcing non-safety security policies with program monitors, in: 10th European Symposium on Research in Computer Security, ESORICS, Milan, 2005.
[17] E. Chang, Z. Manna, A. Pnueli, The safety-progress classification, in: F. Bauer, W. Brauer, H. Schwichtenberg (Eds.), Logic and Algebra of Specifications, Springer-Verlag, 1991, pp. 143–202.
[18] G. Zhu, A. Tyagi, P. Roop, Stream automata as run-time monitors for open system security policies, Department of Electrical and Computer Engineering, Iowa State University, Ames, Iowa, USA Tech Report 06-101, 2006.
[19] J. Ligatti, S. Reddy, A theory of runtime enforcement, with results, in: Proceedings of the European Symposium on Research in Computer Security, ESORICS, 2010.
[20] P. Jun, C. Xingyuan, W. Bei, D. Xiangdong, W. Yongliang, Policy monitoring and a finite state automata model, in: CSSE'08: Proceedings of the 2008 International Conference on Computer Science and Software Engineering, IEEE Computer Society, Washington, DC, USA, 2008, pp. 646–649.
[21] G.H. Mealy, A method for synthesizing sequential circuits, Bell System Technical Journal 34 (5) (1955) 1045–1079.
[22] H. Chabot, R. Khoury, N. Tawbi, Extending the enforcement power of truncation monitors using static analysis, Computers & Security 30 (4) (2011) 194–207.
[23] M. Kim, S. Kannan, I. Lee, O. Sokolsky, M. Viswanathan, Computational analysis of run-time monitoring-fundamentals of Java-mac, Electronic Notes in Theoretical Computer Science 70 (4) (2002).
[24] M. Viswanathan, Foundations for the run-time analysis of software systems, Ph.D. Thesis, University of Pennsylvania, 2000.
[25] P. Fong, Access control by tracking shallow execution history, in: Proceedings of the 2004 IEEE Symposium on Security and Privacy, Oakland, California, USA, May 2004.
[26] T.Y. Lin, Chinese wall security policy—an aggressive model, in: Proceedings of the Fifth Aerospace Computer Security Application Conference, 1989.
[27] K.J. Biba, Integrity considerations for secure computer systems, Tech. Rep., MITRE Corporation, 04, 1977.
[28] G. Edjlali, A. Acharya, V. Chaudhary, History-based access control for mobile code, in: ACM Conference on Computer and Communications Security, 1998, pp. 38–48.
[29] W.E. Boebert, R.Y. Kain, A practical alternative to hierarchical integrity policies, in: Proceedings of the 8th National Computer Security Conference, 1985.
[30] W. Young, P. Telega, W. Boebert, A verified labeler for the secure ADA target, in: Proceedings of the 9th National Computer Security Conference, 1986.
[31] C. Talhi, N. Tawbi, M. Debbabi, Execution monitoring enforcement for limited-memory systems, in: Proceedings of the PST06 Conference, Privacy, Security, Trust, 2006.
[32] C. Talhi, N. Tawbi, M. Debbabi, Execution monitoring enforcement under memory-limitation constraints, in: Proceedings of FCS-ARSPA-06, Joint Workshop on Foundations of Computer Security and Automated Reasoning for Security Protocol Analysis, Associated with FLOC 2006, Federated Logic Conference, 2006.
[33] D. Beauquier, J.-E. Pin, Languages and scanners, Theoretical Computer Science 84 (1) (1991) 3–21.
[34] O. Kupferman, Y. Lustig, M. Vardi, On locally checkable properties, in: Proccedings of the 13th International Conference on Logic for Programming Artificial Intelligence and Reasoning, in: Lecture Notes in Computer Science, Springer-Verlag, 2006.
[35] A.N. Trahtman, An algorithm to verify local threshold testability of deterministic finite automata, in: Lecture Notes in Computer Science, 1999.
[36] A. Magnaghi, H. Tanaka, An efficient algorithm for order evaluation of strict locally testable languages, in: International Conference on Combinatorics, Graph Theroy and Computing, Florida, USA.
[37] S.M. Kim, R. McNaughton, R. McCloskey, A polynomial time algorithm for the local testability problem of deterministic finite automata, IEEE Transactions on Computers 40 (10) (1991) 1087–1093.
[38] S.M. Kim, R. McNaughton, Computing the order of a locally testable automaton, SIAM Journal on Computing 23 (6) (1994) 1193–1215.
[39] D. Perrin, J.E. Pin, Infinite Words, in: Pure and Applied Mathematics, vol. 141, Elsevier, 2004.
[40] D. Beauquier, J. Cohen, R. Lanotte, Security policies enforcement using finite edit automata, Electronic Notes in Theoretical Computer Science 229 (3) (2009) 19–35.
[41] N. Bielova, F. Massacci, Do you really mean what you actually enforced? in: Formal Aspects in Security and Trust: 5th International Workshop, FAST 2008 Malaga, Spain, October 9–10, 2008 Revised Selected Papers, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 287–301.
[42] N. Bielova, F. Massacci, Do you really mean what you actually enforced?-edited automata revisited, International Journal of Information Security 10 (4) (2011) 239–254.
[43] R. Khoury, N. Tawbi, Using equivalence relations for corrective enforcement of security policies, in: The Fifth International Conference Mathematical Methods, Models, and Architectures for Computer Networks Security, 2010.
[44] R. Khoury, N. Tawbi, Corrective enforcement of security policies, in: The 7th International Workshop on Formal Aspects of Security & Trust, FAST2010, 2010.
[45] N. Bielova, F. Massacci, Predictability of enforcement, in: ESSoS, 2011, pp. 73–86.
[46] N. Bielova, F. Massacci, Computer-aided generation of enforcement mechanisms for error-tolerant policies, Pisa, Italy, 6–8 June 2011, in: POLICY 2011, IEEE International Symposium on Policies for Distributed Systems and Networks, 2011, pp. 89–96.
[47] M. Sridhar, K.W. Hamlen, Flexible in-lined reference monitor certification: challenges and future directions, in: Proceedings of the 5th ACM Workshop on Programming Languages Meets Program Verification, PLPV'11, ACM, New York, NY, USA, 2011, pp. 55–60.