Corrective Enforcement of Security Policies

Raphael Khoury and Nadia Tawbi

Laval University, Department of Computer Science and Software Engineering,
raphael.khoury.1@ulaval.ca, nadia.tawbi@ift.ulaval.ca

Abstract. Monitoring is a powerful security policy enforcement paradigm that allows the execution of a potentially malicious software by observing and transforming it, thus ensuring its compliance with a user-defined security policy. Yet some restrictions must be imposed on the monitor's ability to transform sequences, so that key elements of the execution's semantics are preserved. An approximation of the sequence is executed rather than an equivalent one. This approximation must preserve the fundamental behaviors intended by the user. In this paper, we propose a framework to express and study such a restriction based on partial orders. The intuition behind our model is that the monitor should be bounded to output a sequence that both respects the desired security property and is preserves the semantics of the input sequence. We give several examples of real-life security policies and propose monitors capable of enforcing these properties. We then turn to the question of comparing several monitors enforcing the same security property. We propose several metrics which can be used to select the best enforcement paradigm for a given property.

Key words: Monitors, Security Policies Enforcement, Program Transformation

1 Introduction

Monitors have become a widely used security policy enforcement tool. They allow an untrusted program to run safely by observing its execution and reacting as needed to prevent a violation of the security property. Part of their flexibility resides in their ability to transform the input sequence. If a potential violation of the security policy is observed, the monitor may react by aborting the execution, or by adding or removing some program action. In fact, the monitor replaces the execution it observes by an alternate execution that respects the desired security policy.

Previous research has shown that monitors which transform the execution they control are much more powerful than monitors that merely abort invalid executions. For this kind of enforcement to be meaningful, constraints must be imposed on the monitor's ability to transform sequences. Otherwise, the monitor can enforce the property by replacing any execution with an arbitrarily chosen valid sequence.

Previously, researchers have attempted to solve this problem by sorting out sequences into equivalence classes and imposing that the monitor's output be equivalent to the original sequence in either some or all cases. In this context, the equivalence relation represents a semantic property of the original sequence that must be preserved throughout any transformation performed by the monitor. Although this approach solves the

problem described above, it also limits the monitor ability to take certain corrective actions, since the equivalence between the monitor's inputs and outputs must be always maintained. Furthermore, it is often difficult to define a suitable equivalence relation.

We propose a new framework to model the corrective capabilities of monitors. Our key insight is to organize executions into partial orders, rather than equivalence classes, and impose that an execution be replaced only by a sequence that is at least as high on the partial order than itself. As we argue in section 4, partial orders are a more natural way to model the restrictions we wish to impose on the monitor than equivalence relations. We also illustrate our framework with four examples of real properties.

The remainder of this paper is organized as follows. Section 2 presents a review of related work. In Section 3, we define some concepts and notations that are used throughout the paper. In Section 4, we show how partial orders form the basis of a corrective monitoring framework, and motivate its use by comparing it to other enforcement paradigms. In Section 5, we give four examples of security properties that we are able to enforce. Concluding remarks and future work are sketched in Section 6.

2 Related work

This paper extends the body of research that seeks to study the notion of security policy enforcement by monitors and to identify the set of properties enforceable by monitors under various constraints.

These issues were first investigated by Schneider in [12], who formalized the notions of monitoring and enforcement. He focused on specific classes of monitors that observe 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. To enforce a property, this particular monitor must accept each action of any valid sequence, as soon as it is produced by the target program, a enforcement paradigm termed precise enforcement. Under these conditions, a monitor can enforce the class of security policies that are identified in the literature as *safety* properties, and are informally characterized by prohibiting a certain bad thing from occurring in a given execution.

In [1], Ligatti, Bauer and Walker show that if this definition of enforcement is used, the added power of some monitors to transform the executions they monitor (by inserting or suppressing program actions) does not result in an increase in the set of enforceable properties. The authors suggest the alternative notion of effective$_{\cong}$ enforcement instead. A monitor effectively$_{\cong}$ enforces a property if any execution respecting the property is replaced by an equivalent execution, w.r.t. some equivalence relation $\cong$. Subsequently, in [10], the authors delineate the set of properties that are effectively$_{\cong}$ enforceable for a specific equivalence relation, syntactic equality.

Khoury et al. [9] tighten this definition by imposing that all sequences, rather than only the valid ones, be transformed into equivalent ones. They show how this enforcement paradigm models reasonable restrictions on monitors ensuring that their enforcement is meaningful. Alternative definitions of enforcement are given in [3] and [11].

The computability constraints that can further restrict a monitor's enforcement power are discussed in [8]. The enforcement power of monitors operating with memory con-

strains is discussed in [7, 14] and [2]; that of monitors relying upon an a priori model of the program's possible behavior is discussed in [1, 6] and [11].

3 Preliminaries

Executions are modeled as sequences of atomic actions taken from a finite or countably infinite set of 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 is noted $\Sigma^\infty = \Sigma^\omega \cup \Sigma^*$. Likewise, for a set of sequences $\mathcal{S}$, $\mathcal{S}^*$ denote 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* there exists a sequence σ' such that $\tau;\sigma' = \sigma$. We write $\tau \prec \sigma$ (resp. $\sigma \succ \tau$) for $\tau \preceq \sigma \wedge \tau \neq \sigma$ (resp. $\sigma \succeq \tau \wedge \tau \neq \sigma$). Finally, let $\tau, \sigma \in \Sigma^\infty$, τ is said to be a suffix of σ iff there exists a $\sigma' \in \Sigma^*$ s.t. $\sigma = \sigma';\tau$.

We denote by $pref(\sigma)$ (resp. $suf(\sigma)$) the set of all prefixes (resp. suffixes) of σ. Let $A \subseteq \Sigma^\infty$ be a set 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 i^{th} action in a sequence σ is given as σ_i, σ_1 denotes the first action σ, $\sigma[i,j]$ denotes the sequence occurring between the i^{th} and j^{th} actions of σ, and $\sigma[i,..]$ denotes the remainder of the sequence, starting from action σ_i. The length of a sequence $\tau \in \Sigma^*$ is given as $|\tau|$. Let τ, τ' be sequences, we write $\tau\backslash\tau'$ for the left cancelation of τ' from τ, which is defined by the truncation from τ of the first occurrence of each occurrence of each action present in τ'. For example, $a;b;c;a;d;a\backslash d;a;a = b;c;a$.

A multiset, or bag is a generalization of a set in which each element may occur multiple times. A multiset $\mathcal{A}$ can be formally defined as a pair $\langle A, f\rangle$ where A is a set and $f : A \to \mathbb{N}$ is a function indicating the number of occurrences of each element of A in $\mathcal{A}$. Note that $a \notin A \Leftrightarrow f(a) = 0$. Thus, by using this insight, to define basic operations on multisets one can consider a universal set A and different functions of type $A \to \mathbb{N}$ associated with it to form different multisets. We let $acts(\tau)$ represent the multiset of actions occurring in sequence τ.

Finally, a security policy $P \subseteq \Sigma^\infty$ is a set of allowed executions. A policy P is a property iff there exists a decidable predicate $\hat{\mathcal{P}}$ over the executions of Σ^∞ s.t. $\sigma \in P \Leftrightarrow \hat{\mathcal{P}}(\sigma)$. Thus, a property is a policy for which the membership of a sequence can be determined solely by examining it. Such a sequence is said to be valid. Since all policies enforceable by monitors are properties, we use $\hat{\mathcal{P}}$ to refer to policies and their characteristic predicate interchangeably.

4 Monitoring with Partial Orders

In this section, we introduce the automata-based model of monitors and the notions of effective$_\cong$ and corrective$_\cong$ enforcement used in the literature. We show why these definitions are sometimes inadequate before presenting an alternative enforcement paradigm.

The edit automaton [1, 10], is the most widely used model of a monitor. It captures the behavior of a monitor capable of inserting or suppressing any action in the execution

in progress, as well as halting it. The ideas presented in this paper are easily transferable to the model proposed by Ligatti et al. in [11].

Definition 1. *An edit automaton is a tuple* $\langle \Sigma, Q, q_0, \delta \rangle$ *where*[1]*:*

- Σ *is a finite or countably infinite set of actions;*
- Q *is a finite or countably infinite set of states;*
- $q_0 \in Q$ *is the initial state;*
- $\delta : (Q \times \Sigma) \rightarrow (Q \times \Sigma^\infty)$ *is the transition function, which, given the current state and input action, specifies the automaton output and successor state. At any step, the automaton may accept the action and output it as it is, suppress it and move on to the next action, (with no output), or output some other sequence in* Σ^∞*. If an undefined transition is attempted, the automaton aborts.*

Let $\mathcal{A}$ be an edit automaton, we let $\mathcal{A}(\sigma)$ be the output of $\mathcal{A}$ when its input is σ.

Most related studies have focused on effective enforcement. A mechanism effectively enforces a security property iff it respects the two following principles, from [1]:

1. $Soundness$: All output must respect the desired property.
2. $Transparency$: The semantics of executions which already respect the property must be preserved. This naturally requires the use of an equivalence relation, stating when one sequence can be substituted for another.

Definition 2. ***(From [1])*** *Let* $\mathcal{A}$ *be an edit automaton.* $\mathcal{A}$ *effectively*$_\cong$ *enforces the property* $\hat{\mathcal{P}}$ *iff* $\forall \sigma \in \Sigma^\infty$

1. $\hat{\mathcal{P}}(\mathcal{A}(\sigma))$ *(i.e.* $\mathcal{A}(\sigma)$ *is valid)*
2. $\hat{\mathcal{P}}(\sigma) \Rightarrow \mathcal{A}(\sigma) \cong \sigma$

In the literature, the only equivalence relation $\cong$ for which the set of effectively$_\cong$ enforceable properties has been formally studied is syntactic equality[1]. Yet, effective enforcement is only one paradigm of enforcement which has been suggested. Other enforcement paradigms include precise enforcement[1], all-or-nothing delayed enforcement[3], conservative enforcement[1] and corrective$_\cong$ enforcement [9].

Most previous work has focused on effective$_\cong$ enforcement. This definition allows the monitor to replace an invalid execution with any valid sequence, even ϵ. A more intuitive model of the desired behavior of a monitor would rather require that only minimal alterations be made to an invalid sequence, for instance by releasing a resource or adding an entry in a log. Those parts of the input sequence which are valid should be preserved in the output, while invalid behaviors should be corrected or removed. A possible solution is proposed by Khoury et al. in [9], and was termed corrective$_\cong$ enforcement. An enforcement mechanism correctively$_\cong$ enforces the desired property if every output sequence is both valid and equivalent to the input sequence. The equivalence relation is formulated such that the valid behavior of the input sequence is preserved.

Definition 3. ***(From[9])*** *Let* $\mathcal{A}$ *be an edit automaton.* $\mathcal{A}$ *correctively*$_\cong$ *enforces the property* $\hat{\mathcal{P}}$ *iff* $\forall \sigma \in \Sigma^\infty$

[1] This definition, taken from [14], is equivalent to the one given in [1].

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

However, this definition also raises some difficulties. In particular, it implies that several distinct valid sequences, which are possible transformations of an invalid sequences, must be equivalent. Likewise, it requires that several invalid sequences be considered equivalent if a single valid sequence is a valid alternative to both.

In this paper, we examine an alternative notion of enforcement, termed corrective$_{\sqsubseteq}$ enforcement. Following previous work in monitoring by Fong [7], we use an abstraction function $\mathcal{F} : \Sigma^* \rightarrow \mathcal{I}$, to capture the property of the input sequence that the monitor must preserve throughout its manipulation. While Fong made use of abstractions to reduce the overhead of the monitor, we use them as the basis for determining which transformations the monitor is or is not allowed to perform on both valid and invalid sequences. The main idea is to use this abstraction to capture the semantic property of the execution corresponding to the valid, or desired behavior of the target program. This may be, for example, the number of occurrences of certain subwords or factors or any other semantic property of interest.

We wish to constrain the behavior of the monitor so that an invalid sequence is corrected in such a way that the monitor preserves all valid behaviors present in it. An intuitive solution to this problem would be to impose that the output sequences always have the same value of $\mathcal{F}$. The abstraction function would thus form the basis of a partition of the sequences of Σ^∞ into equivalence classes. But, as discussed above, this limits the monitor's ability to use the same valid sequences as a potential solution to several unrelated invalid sequences. Instead, we let $\sqsubseteq$ be a partial order over sequences of Σ^*, s.t. $\forall \sigma, \sigma' : \sigma \sqsubseteq \sigma' \Leftrightarrow \mathcal{F}(\sigma) \leq \mathcal{F}(\sigma')$. The monitor is allowed to transform an input σ into another sequence σ' iff $\sigma \sqsubseteq \sigma'$. By defining $\sqsubseteq$ appropriately, we can ensure that sequences which are greater on the partial order are always adequate replacements for any inferior sequences, in the sense that they preserve the valid behaviors present in those sequences. The use of partial order also allows us to connect monitoring to refinement specifications, which are stated using partial orders. We also find that partial orders are a more natural way to state most security policies than equivalence relations. Let τ, τ' be two sequences s.t. $\tau \sqsubseteq \tau'$, we write that τ is lower than τ' or conversely, that τ' is higher than τ.

Definition 4. *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 $\hat{\mathcal{P}}$ iff $\forall \sigma \in \Sigma^\infty$*

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

A monitor often operates in a context in which it knows that certain executions cannot occur. This is because the monitor can benefit from a static analysis of its target, that provides it with a model of the target's possible behavior. Prior research [1, 6] has shown that a monitor operating in such a context (called a *nonuniform* context) can enforce a significantly larger range of properties than one that considers every sequence in Σ^∞ to be a possible input (called the *uniform* context). To take into account the possibility that the monitor might operate in a nonuniform context, we adapt the preceding definition as follows:

Definition 5. *Let $\mathcal{S} \subseteq \Sigma^\infty$ be a subset of sequences, let $\sqsubseteq$ be a partial order over the sequences of Σ^∞ and let $\mathcal{A}$ be an edit automaton. $\mathcal{A}$ correctively$^{\mathcal{S}}_{\sqsubseteq}$ enforces the property $\hat{\mathcal{P}}$ iff $\forall \sigma \in \mathcal{S}$*

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

We write corrective$_{\sqsubseteq}$ enforcement when $\mathcal{S} = \Sigma^\infty$ or $\mathcal{S}$ is obvious from context.

The partial order is defined over the sequences of Σ^*. Let σ, σ' be infinite sequences, we have that $\sigma \sqsubseteq \sigma'$ iff σ and σ' have infinitely many common prefixes τ, τ' s.t. $\tau \sqsubseteq \tau'$ with $\tau \prec \sigma$ and $\tau' \prec \sigma'$.

$$\forall \sigma, \sigma' \in \Sigma^\omega : \sigma \sqsubseteq \sigma' \Leftrightarrow \forall \tau \prec \sigma : \exists \upsilon \succeq \tau : \exists \tau' \prec \sigma' : \upsilon \sqsubseteq \tau' \tag{4.1}$$

Finally, since the monitor operates by transforming sequences, we must impose that every partial order respects the following closure restriction.

$$\tau \sqsubseteq \tau' \Rightarrow \tau; \sigma \sqsubseteq \tau'; \sigma \tag{4.2}$$

To understand the need for this restriction consider the possible behavior of a monitor which is presented with an invalid prefix τ of a longer input sequence. It may opt to transform τ into a valid higher sequence τ'. However, if the closure restriction given in equation 4.2 is not respected by the partial order, then it's possible that the full input sequence $\sigma \succ \tau$ is actually valid, but that there is no valid extension of τ' that is greater than σ. The monitor would have inadvertently ruined a valid sequence.

5 Examples

In this section, we illustrate the use of corrective$_{\sqsubseteq}$ enforcement using four real-life security properties : transactional properties, the assured pipeline property, the Chinese wall property and general availability.

5.1 Transactional Properties

The first class of properties we wish to correctively$_{\sqsubseteq}$ enforce is that of transactional properties, suggested in [10]. Let Σ be an action set and let $\mathcal{T} \subseteq \Sigma^*$ be a set of finite transactions, $\hat{\mathcal{P}}_{\mathcal{T}}$ is a *transactional* property over set Σ^∞ iff

$$\forall \sigma \in \Sigma^\infty : \hat{\mathcal{P}}_{\mathcal{T}}(\sigma) \Leftrightarrow \sigma \in \mathcal{T}^\infty \tag{transactional}$$

This definition is subtly different, and indeed forms a subset of that of iterative properties defined in [3]. Transactional properties also form a subset of the set of *renewal* properties, and include some but not all *safety* properties, *liveness* properties as well as properties which are neither.

Transactional properties can be effectively$_{=}$ enforced [10], in a manner that allows the longest valid prefix to be output [2]. In [3], Bielova et al. propose an alternative

enforcement paradigm, that allows all valid transactions to be output. Corrective$_{\sqsubseteq}$ enforcement is a generalization of their work.

The partial order which we will consider is based on the notion of factors. A word $\tau \in \Sigma^*$ is a factor of a word $\omega \in \Sigma^\infty$ if $\omega = \upsilon;\tau;\upsilon'$, with $\upsilon \in \Sigma^*$ and $\upsilon' \in \Sigma^\infty$. Factors allow us to reason about transactions in a formalized manner.

We will only consider transactional properties built from a set of sequences $\mathcal{T}$ which meets the unambiguity criterion suggested in [9].

$$\forall \sigma, \sigma' \in \mathcal{T} : \forall \tau \in pref(\sigma) : \forall \tau' \in suf(\sigma') : \tau \neq \epsilon \wedge \tau' \neq \epsilon \Rightarrow \tau;\tau' \notin \mathcal{T} \quad \text{(unambiguity)}$$

Informally, this restriction forbids the occurrence of transactions with overlapping prefixes and suffixes. If this restriction is not met, there exists sequences which cannot be parsed as the concatenation of valid transactions, and thus not in the property, but for which every action is a part of at least one valid transaction (with some atomic actions belonging to more than one transaction).

To enforce this property, we use an abstraction function which returns the multiset of factors occurring in a given sequence. We use a multiset rather than simply comparing the set of factors from $\mathcal{T}$ occurring in each sequence so as to be able to distinguish between sequences containing a different number of occurrences of the same subset of factors. For any two sequences σ and σ', we write $\sigma \sqsubseteq \sigma'$ iff σ' has more valid factors (w.r.t. $\mathcal{T}$) than σ, and σ' is thus an acceptable replacement sequence for σ, provided that the σ' is valid and σ is not. This captures the intuition that if certain valid transactions are present in the input sequence, they must still be present in the output sequence, regardless of any other transformation made to ensure compliance with the security property. The monitor may add valid transactions and remove invalid ones, but may not remove any valid transactions present in the original execution.

Let $valid_{\mathcal{T}}(\sigma)$, which stand for the multiset of factors from the sequence σ which are present in $\mathcal{T}$, be the abstraction function $\mathcal{F}$. The partial order $\sqsubseteq$ used to correctively enforce this property is thus given as $\forall \sigma, \sigma' \in \Sigma^\infty : \sigma \sqsubseteq \sigma' \Leftrightarrow valid_{\mathcal{T}}(\sigma) \subseteq valid_{\mathcal{T}}(\sigma')$. This partial order captures the intuition that any valid transaction present in the original sequence must also be present in the monitor's output.

The following automaton correctively enforces transactional properties. Let $\mathcal{A}_t = \langle \Sigma, Q, q_0, \delta \rangle$ where

- Σ is a finite or countably infinite action set.
- $Q = \Sigma^* \times \Sigma^*$, is the set of automaton states. Each state consists of a pair $\langle \sigma_o, \sigma_s \rangle$ where σ_o is the sequence which has been output so far, and σ_s is a sequence which the monitor has suppressed, and may either eventually output or delete.
- $q_0 = \langle \epsilon, \epsilon \rangle$, is the initial state.
- The transition function $\delta : Q \times \Sigma \to Q \times \Sigma^\infty$ is given as:

$$\delta(\langle \sigma_o, \sigma_s \rangle, a) = \begin{cases} \langle \sigma_o;\tau, \epsilon \rangle & \text{if } \exists \tau \in suf(\sigma_s;a) : \tau \in \mathcal{T} \wedge \tau \neq \epsilon \wedge |\tau| \leq |\sigma_s;a| \\ \langle \sigma_o;\sigma_s;a, \epsilon \rangle & \text{if } \exists \tau \in \mathcal{T} : \exists \tau' \in \mathcal{T}^* : \sigma_o;\sigma_s;a = \tau';\tau \wedge \\ & \quad |\tau| \geq |\sigma_s;a| \\ \langle \sigma_o, \sigma_s;a \rangle & \text{otherwise} \end{cases}$$

Proposition 1. *Let $\mathcal{T} \subseteq \Sigma^\infty$ be a subset of sequences, let $\hat{\mathcal{P}}_{\mathcal{T}}$ be the corresponding transactional property and let $\sqsubseteq$ be a partial order over the sequences of Σ^∞ defined*

such that $\forall \sigma, \sigma' \in \Sigma^\infty : \sigma \sqsubseteq \sigma' \Leftrightarrow valid_{\mathcal{T}}(\sigma) \subseteq valid_{\mathcal{T}}(\sigma')$. *The automaton* $\mathcal{A}_t$ *correctively*$_{\sqsubseteq}$ *enforces* $\hat{\mathcal{P}}_{\mathcal{T}}$.

Proof. We omit the proofs of theorems and propositions due to space constraints. □

The method above thus shows how a transactional property could be enforced in such a manner that the output is always valid, and always contains as many or more valid transactions than the input. In this particular example, we may be able to prove an even stronger enforcement paradigm, namely that the output will always contain exactly the same valid transactions as the input. This is unsurprising, since equivalence relations are a special case of partial orders, and imposing such a constraint would be tantamount to using the equivalent$_{\cong}$ enforcement proposed in section 4. The method presented here can thus be seen as a generalization of this framework.

But this example also highlights why equivalent$_{\cong}$ enforcement is too rigid to be useful in many practical cases. Consider what would happen, for example, if the restriction that the set $\mathcal{T}$ be unambiguous is lifted [2]. Even though the only transformation performed by the monitor is to remove invalid factors, we cannot guarantee that exactly those valid sequences which are present in the original sequence will be present in the output. As a counterexample, consider the case of valid sequences $\mathcal{T} = \{\sigma_1, \sigma_2, \sigma_3\}$ and invalid sequences $\tau, \tau' \notin \mathcal{T}$ s.t. $\tau; \sigma_3; \tau' = \sigma_1; \sigma_2$. Let the infinite sequences $\sigma_1; \upsilon; \sigma_2; \upsilon; \sigma_1; \upsilon ...$ be the input sequence, where υ is an invalid transaction (possibly τ or τ'). The multiset of valid transactions present in $\sigma_1; \sigma_2, \sigma_1; \sigma_2, \sigma_1; \sigma_2 ...$ contains an infinite number of factors σ_3, not present in the original sequence. While it may be difficult to imagine a real-life, transactional property exhibiting this behavior, this example does illustrate why using equivalence relations rather than a partial order would unduly restrict the transformations available to a monitor. Furthermore, one may wish to consider other means by which a monitor may enforce a transactional property, for instance, by inserting actions to correct an incomplete or transactions, or simply to log the occurrence of certain possibly malicious factors.

5.2 Assured Pipelines

In the previous section, we show how transactional properties can be enforced by an edit automaton that simply suppresses some actions from the input. Yet, part of the power of the edit automaton resides in its ability to insert actions not present in the input to correct an invalid sequence. Naturally, this ability must be constrained for the enforcement to remain meaningful, otherwise the monitor may simply replace any invalid sequence by some unrelated valid sequence. In this section, we propose two possible enforcement paradigms for the assured Pipeline policy based on suppression and insertion.

The assured pipeline property was suggested in [4] to ensure that data transformations are performed in a specific order. Let O be a set of data objects, and S be a set of transformations. We assume that S contains a distinguished member **create**. Finally, let $E \in \langle S \times O \rangle$ be a set of access events. $\langle s, o \rangle \in E$ denotes the application of transformation s to the data object o. An assured pipeline policy restricts the application of transformations from S to data objects using an enabling relation $e : S \times S$, with the

[2] This may involve altering the definition of iterative properties.

following two restrictions: the relation e must define an acyclic graph, and the **create** process can only occur at the root of this graph. The presence of a pair $\langle s, s' \rangle$ in e, is represented in the graph by the occurrence of an edge between the vertex s and the vertice s', and indicates that any action of the form $\langle s', o \rangle$ is only permissible if s is the last process that accessed o. Because of the restriction that e must be a acyclic graph, each action $\langle s', o \rangle$ can occur at most once during an execution.

We add another restriction namely that the enabling relation be linear. Formally : $\forall s, s' \in S : \langle s, s' \rangle \in e \Rightarrow \neg \exists \langle s, s'' \rangle \in e : s' \neq s''$. This condition will make it easier for insertion monitors to add actions to the output, without compromising transparency.

A truncation-based monitor was suggested by Fong [7] and Talhi [14] to enforce this property. Their monitors effectively$_{=}$ enforce the assured pipeline property by aborting the execution if an unauthorized data transaction is encountered. Our corrective enforcement framework allows the monitor to continue the execution after an illicit transformation has been attempted.

As was the case with transactional properties, the partial order defining the desired behavior of the program is stated in terms of the presence of valid actions, i.e. those occurring in a manner allowed by the enabling relation. Since each action can only occur once, a function returning the set of valid atomic actions is an adequate abstraction function. We write $valid_e(\sigma)$ for the set of valid transformations (w.r.t. enabling relation e) occurring in σ. We write $\sigma \sqsubseteq \sigma' \Leftrightarrow valid_e(\sigma) \subseteq valid_e(\sigma')$.

Instead of simply aborting the execution, a corrective monitor may suppress an invalid action, and allow the execution to proceed. The automaton $\mathcal{A}^s_{ap} = \langle E, Q, q_0, \delta_s \rangle$ bellow which enforces the assured pipelines in this manner.

- E is a set of access events over objects from O and transformations from S.
- $Q : \wp(E)$ is the state space. Each state is an unordered set of access events from E.
- $q_0 = \emptyset$ is the initial state.
- The transition function $\delta_s : Q \times \Sigma \rightarrow Q \times \Sigma^\infty$ is given as:

$$\delta_s(q, \langle s, o \rangle) = \begin{cases} (q \cup \{\langle s, o \rangle\}, \langle s, o \rangle) & \text{if } s = create \wedge \langle create, o \rangle \notin q \\ (q \cup \{\langle s, o \rangle\}, \langle s, o \rangle) & \text{if } \exists s' \in S : \langle s', o \rangle \in q \wedge \langle s', s \rangle \in e \wedge \\ & \quad \langle s, o \rangle \notin q \\ (q, \epsilon) & \text{otherwise} \end{cases}$$

Proposition 2. *Let e be an enabling relation, defining an assure pipeline policy $\hat{\mathcal{P}}$ over a set of actions E, and let $\sqsubseteq$ be a partial order over the sequences of E^∞ defined such that $\forall \sigma, \sigma' \in E^\infty : \sigma \sqsubseteq \sigma' \Leftrightarrow valid_e(\sigma) \subseteq valid_e(\sigma')$. The automaton $\mathcal{A}^s_{ap}$ correctively$_{\sqsubseteq}$ enforces $\hat{\mathcal{P}}$.*

The execution thus either outputs an action if it is valid, or it suppresses it before allowing the execution to continue. Yet, the edit automaton is capable of not only suppressing or outputting the actions present in the input, but also of adding actions not present in the execution to the input. It may be reasonable, for instance, for a monitor to suppress only those transformations which have already occurred, or those manipulating an object which has not yet been created. Otherwise, if an action occurs in the input sequence before the actions preceding it in e have occurred, the monitor can enforce the property by adding the required actions. The following automaton $\mathcal{A}^e_{ap}$ enforces the property as described above.

To simplify the notation, we use the predicate $path_e(\tau)$ to indicate that τ is a factor of a valid sequence according to e and that every action in τ manipulates the same object. Formally , $path_e(\tau) \Leftrightarrow$

- $\exists o \in O : \forall \langle s, o' \rangle \in acts(\tau) : o' = o$
- $\forall i : 2 \leq i \leq |\tau| : \tau_{i-1} = \langle s, o \rangle \wedge \tau_i = \langle s', o \rangle \Rightarrow \langle s, s' \rangle \in e$

Let $\mathcal{A}^e_{ap} = \langle E, Q, q_0, \delta_e \rangle$ where Σ, Q and q_0 are defined as in $\mathcal{A}^s_{ap}$ and $\delta_e$is given as follows.

$$\delta_e(q, \langle s, o \rangle) = \begin{cases} (q \cup \{\langle s, o \rangle\}, \langle s, o \rangle) & \text{if } s = create \wedge \langle create, o \rangle \notin q \\ (q \cup \{\langle s, o \rangle\}, \langle s, o \rangle) & \text{if } \exists s' \in S : \langle s', o \rangle \in q \wedge \langle s', s \rangle \in e \wedge \\ & \langle s, o \rangle \notin q \\ (q \cup acts(\tau; \langle s, o \rangle), \tau; \langle s, o \rangle) & \text{if } \langle s, o \rangle \notin q \wedge \langle create, o \rangle \in q \\ & \text{and } \tau \text{ is the longest sequence s.t.} \\ & \tau_0 \notin q \wedge path(\tau; \langle s, o \rangle) \\ (q, \epsilon) & \text{otherwise} \end{cases}$$

Proposition 3. *Let e be an enabling relation, defining an assured pipeline policy $\hat{\mathcal{P}}$ over a set of actions E, and let $\sqsubseteq$ be a partial order over the sequences of E^∞ defined such that $\forall \sigma, \sigma' \in E^\infty : \sigma \sqsubseteq \sigma' \Leftrightarrow valid_e(\sigma) \subseteq valid_e(\sigma')$. The automaton $\mathcal{A}^e_{ap}$ correctively$_\sqsubseteq$ enforces $\hat{\mathcal{P}}$.*

Note that had we used an equivalence relation rather than a partial order, it would not have been possible to correct the sequence in this manner as this requires the input sequence be equivalent a corrected sequence to which valid actions have been added. This in turn implies that whenever such actions occur in an input sequence, the monitor is allowed to delete them, which is obviously not desirable. Partial orders allow the monitor to alter the original sequence so as to correct violations of the security policies, while preserving key elements of the input sequence's semantics.

5.3 Chinese Wall

The third example we consider is the Chinese Wall policy, suggested in [5] to avoid conflicts of interest. In this model, a user which accesses a data object o is forbidden from accessing other data objects that are in conflict with o. Several implementations of this model have been suggested in the literature. In this paper, we consider the framework of Sobel et al. [13], which includes the notion of data relinquishing.

Let S be a set of subjects, and O a set of objects. The set of conflicts of interests is given as a set of pairs $C : O \times O$. The presence of $(o_i, o_j) \in C$ indicates that objects o_i and o_j conflict. Naturally, C is a symmetric set ($(o_i, o_j) \in C \Leftrightarrow (o_j, o_i) \in C$). For all objects $o_i \in O$, we write C_{o_i} for the set of objects which are in conflict with o_i. Let $O' \subseteq O$ be a subset of objects, overloading the notation we write $C_{O'}$ for $\bigcup_{o \in O'} C_o$. An action of the form ($\mathbf{acq}$, s, o) indicates that subject s holds the right to acquires the right to access object o. After this action is performed the subject can freely access the resource but can no longer access an object which conflicts with o.

It can often be too restrictive to impose on subjects that they never again access any data that conflict with any data objects they have previously accessed. In practice,

the involvement of a subject with a given entity will eventually come to an end. Once this occurs, the subject should no longer be prevented from collaborating with the competitors of his former client. In [13], the model is enriched along this line by giving subjects the capacity to relinquish previously acquired data. This can be modeled by the action (rel, s, o), indicating that subject s relinquishes a previously accessed object o. After this action occurs in a sequence, s is once again allowed to access objects that conflict with o, as long as they do not conflict with any other objects previously accessed by s and not yet relinquished. To simplify the notation, we define function $live : S \times \Sigma^* \rightarrow \wp(O)$ as follows : let s be a subject and τ be a finite sequence, $live(s, \tau)$ return the set of objects which s has accessed in τ and has not yet released.

The security property predicate is stated as follows: $\forall \sigma \in \Sigma^\infty : \hat{\mathcal{P}}(\sigma) \Leftrightarrow \forall s \in S : \forall o \in O : \forall i \in N : \sigma_i = (\textbf{access}\ , s, o) : \neg \exists o' \in live(s, \sigma[..i-1]) : o \in C_{o'}$ where $\Sigma = \{\textbf{access,rel}\} \times S \times O$ is the set of atomic actions.

Both Fong [7] and Talhi [14] enforce this property by truncation, aborting the execution as soon as a conflicting data access is attempted. Any authorized data access present in the input sequence after a conflicting data access has occurred is absent from the output sequence. Corrective enforcement can provide a more flexible enforcement. First, we define the partial order $\sqsubseteq$. Analogously to section 5.1, we impose that authorized data accesses are preserved while conflicting ones are deleted. Let σ be a sequence and let C be the corresponding conflict of interest class, we write $valid_C(\sigma)$ for the multiset of actions from σ occurring in that sequence in a manner consistent with C. The partial order is defined as $\forall \sigma, \sigma' \in \Sigma^\infty : \sigma \sqsubseteq \sigma' \Leftrightarrow valid_C(\sigma) \sqsubseteq valid_C(\sigma')$. The most intuitive manner to correctively$_\sqsubseteq$ enforce this property is to suppress conflicting data access, but allow the execution to proceed afterward. The following automaton enforces the property the Chinese wall property is this manner.

$\mathcal{A}^s_{cw} = \langle E, Q, q_0, \delta_s \rangle$ where

- $\Sigma : \{\textbf{access,rel}\} \times S \times O$ is the set of all possible access and release events over objects from O and subjects from S,
- $Q : \Sigma^*$ is the state space. Each state is the finite sequence which has been output so far.
- $q_0 = \epsilon$ is the initial state.
- $\delta_s : Q \times \Sigma \rightarrow Q \times \Sigma \cup \{\epsilon\} \times \Sigma^\infty$ is given as :
- $\delta_s(q, a) = \begin{cases} (q; a, a) \text{ if } a = (\textbf{access}\ , s, o)) \wedge o \notin live(s, q) \\ (q; a, a) \text{ if } a = (\textbf{rel}\ , s, o) \\ (q, \epsilon) \quad \text{otherwise} \end{cases}$

Proposition 4. *Let C be a set of conflicts of interests, and let $\hat{\mathcal{P}}_C$ be the corresponding Chinese Wall property. The automaton $\mathcal{A}^s_{ap}$ correctively$_\sqsubseteq$ enforces $\hat{\mathcal{P}}_C$.*

As discussed above, the involvement of any subject with a given entity eventually ends. When this occurs, objects in conflict with that entity become available to this subject for access, provided that they do not conflict with any other object currently held by this subject. Thus, it is natural to suggest an enforcement paradigm in which the monitor keeps track of denied data accesses, and inserts them after release actions make them available. In fact, access to conflicting objects is delayed rather than suppressed. Since the monitor must return an output which is superior or equal to the input on

a partial order, rather than in the same equivalence class, the monitor is allowed to transform the input sequence such that it contains strictly more valid accesses.

The automaton $\mathcal{A}^e_{cw}$ enforces the property in this manner.
$\mathcal{A}^e_{cw} = \langle E, Q, q_0, \delta_e \rangle$ where

- $\Sigma : \{access, rel\} \times S \times O$ is the set of all possible access and release events over objects from O and subjects from S,
- $Q : \Sigma^* \times \Sigma^*$ is the state space. Each state is a pair $\langle \sigma_o, \sigma_s \rangle$ of finite sequences, where σ_o is the sequence which has been output so far, and σ_s is the sequence which has been seen and suppressed.
- $q_0 = \langle \epsilon, \epsilon \rangle$ is the initial state.
- $\delta_e : Q \times \Sigma \to Q \times \Sigma \cup \{\epsilon\} \times \Sigma^\infty$ is given as :

$$\delta_e(\langle \sigma_o, \sigma_s \rangle, a) = \begin{cases} (\langle \sigma_o; a, \sigma_s \tau \rangle) & \text{if } a = (\textbf{access}\ , s, o)) \wedge o \notin live(s, \sigma_o) \\ (\langle \sigma_o; a; \tau, \sigma_s \backslash \tau \rangle) & \text{if } a = (\textbf{rel}\ , s, o) \rangle \text{and } f(s, \sigma_o, \sigma_s) = \tau \\ (\langle \sigma_o, \sigma_s; a \rangle) & \text{otherwise} \end{cases}$$

where the function f examines the sequences have been suppressed and output up to that point and returns a sequence composed of all actions which have previously been suppressed, but can now be output with causing a conflict.

Proposition 5. *Let C be a set of conflicts of interests, and let $\hat{\mathcal{P}}_C$ be the corresponding Chinese Wall property. The automaton $\mathcal{A}^e_{ap}$ correctively$_\sqsubseteq$ enforces $\hat{\mathcal{P}}_C$.*

5.4 General Availability

The final security property that we examine is general availability; a policy requiring that any acquired resource is eventually released. Despite its apparent simplicity, the property is remarkably difficult to monitor in a system with more than one resource, and is given by Ligatti et al. as an example of a property that is not effectively$_=$ enforceable.

Modifying Ligatti's formulation slightly, we define the property as follows. Let $(ac, i), (use, i)$ and (rel, i) stand for accessing, using and releasing a resource i from a set of resources $\mathcal{I}$. The set of possible actions is given as $\Sigma : \{ac, use, rel\} \times \mathcal{I}$. The property states that only acquired resources are used, and that any acquired resource is eventually released. This seemingly straightforward property combines a liveness component, that cannot be monitored [10], with a safety component.

In related work, there has been an interest in examining how restricting the set of possible executions can increase that of enforceable properties. The intuition is that if the monitor knows, from a static analysis, that certain executions cannot occur, it should be able to enforce properties that would be otherwise unenforceable. This question was first raised in [12], and was later addressed in [1, 6, 11].

It this section, we show that an a priori knowledge of the program's possible executions not only increases the set of properties enforceable by the monitor, but also provides a different, and possibly preferable, enforcement of a given property. A monitor trying to enforce the general availability property when $\mathcal{S} = \Sigma^\infty$ cannot simply abort an invalid execution since it could be corrected. Nor can it suppress a potentially invalid sequence only to output it when a valid prefix is reached. As observed in [10], an infinite sequence of the form $(ac, 1); (ac, 2); (rel, 1); (ac, 3); (rel, 2); (ac, 4); (rel, 3)...$

is valid but has no valid prefix. The property can only be enforced by a monitor transforming the input more aggressively.

Since the desired behavior of the program is given in terms of the presence of (use, i) actions bracket by ac_i and (rel, i) actions, a partial order based on the number of occurrences of such actions is a natural way to compare sequences. We can designate such use actions as valid. Let the function $valid(\sigma)$, which returns the multiset of actions (use, i) which occur in sequence σ and are both preceded by a (ac, i) action and followed by a (rel, i) action be the abstraction function $\mathcal{F}$. We thus have that $\forall \sigma, \sigma' \in \Sigma^* : \sigma \sqsubseteq \sigma' \Leftrightarrow valid(\sigma) \leq valid(\sigma')$.

A trivial way to enforce this property is for the monitor to insert an (ac, i) action prior to each (use, i) action, and follow it with a (rel, i) action. The (ac, i) and rel_i actions present in the original sequence can than simply be suppressed, as every (use, i) action is made available by a pair of actions (ac, i) and (rel, i) inserted by the monitor. While theoretically feasible, it is obvious that a monitor which opens and closes a resource after every program step would be of limited use in practice. Furthermore, for many applications, a sequence of the form $(ac, i); (use, i); (use, i); (rel, i)$ cannot be considered equivalent to the sequence $(ac, i); (use, i); (rel, i); (ac, i); (use, i); (rel, i)$. We thus limit this study to monitors which insert (ac, i) actions a finite number of times.

We first propose a monitor capable of enforcing the general availability property in a uniform context, where every sequence in Σ^∞ can occur in the target program. A monitor can enforce the property in this context by suppressing every acquire action, as well as subsequent use actions for the same resource, and output them only when a release action is reached. Any use action not preceded by a corresponding acquire action is simply suppressed and never inserted again. A monitor enforcing the property in this manner needs to keep track only of the actions that have been suppressed and not output so far. The automaton $\mathcal{A}_{ga}^{\Sigma^\infty}$ enforces the property as described above.

Let $\mathcal{A}_{ga}^{\Sigma^\infty} = \langle \Sigma, Q, q_0, \delta_e \rangle$ where

- $\Sigma : \{aq, rel, use\} \times \mathcal{I}$ is the set of all possible acquire, release and use actions for all objects;
- $Q : \Sigma^*$ is the sequence which has been suppressed so far;
- $q_0 = \epsilon$ is the initial state;
- $\delta_{\Sigma^\infty} : Q \times \Sigma \to Q \times \Sigma \cup \{\epsilon\}$ is given as:

$$\delta_e(\tau, a) = \begin{cases} (\tau, \epsilon) & \text{if } a = (use, i) \wedge (ac, i) \notin acts(\tau) \\ (\tau; a, \epsilon) & \text{if } a = (ac, i) \vee (a = (use, i) \wedge \\ & \quad (ac, i) \in acts(\tau)) \\ (\tau \backslash (f_i(\tau)), f_i(\tau); (rel, i)) & \text{if } a = (rel, i) \end{cases}$$

The purpose of the function f_i is simply to examine the suppressed sequence and retrieve the actions over resource i.

Proposition 6. *The automaton $\mathcal{A}_{ga}^{\Sigma^\infty}$ correctively$_\sqsubseteq$ enforces the general availability property.*

A static analysis can often determine that all computations of a program are *fair* meaning certain actions must occur infinitely often in infinite paths. For the purposes of the general availability property, a static analysis could determine that any action (ac, i)

is eventually followed by a (rel, i) action, or by the end of the execution. The property can thus be violated only by the presence of use actions that are not preceded by a corresponding ac action. If the monitor is operating in a fair context, a new, more conservative enforcement method becomes possible, as is illustrated by automaton $\mathcal{A}_{ga}^{fair}$.

Let $\mathcal{A}_{ga}^{fair} = \langle \Sigma, Q, q_0, \delta_e \rangle$ where Σ is defined as above and

- $Q : \wp(\mathcal{I})$ is the set of resources which have been acquired but not yet released;
- $q_0 = \emptyset$ is the initial state;
- $\delta_{\Sigma^\infty} : Q \times \Sigma \to Q \times \{\Sigma \cup \epsilon\}$ is given as :

$$\delta_e(q,a) = \begin{cases} (q, \epsilon) & \text{if } a = (use, i) \wedge (ac, i) \notin acts(\tau) \\ (q \cup \{i\}, a) & \text{if } a = (ac, i) \\ (q \backslash i, a) & \text{if } a = (rel, i) \\ (q, a) & \text{if } a = (use, i) \wedge i \in q \\ (\epsilon, f(q)) & \text{if } a = a_{end} \end{cases}$$

Where $f : \wp(\mathcal{I}) \to \Sigma^*$ is a function returning a sequence of the form $rel_i, rel_j, rel_k ...$ for all resources present in its input.

Proposition 7. *Let $\mathcal{S} \subseteq \Sigma^\infty$ be a subset of sequences s.t. $\forall \sigma \in \mathcal{S} \cap \Sigma^\omega : \forall i \in \mathcal{I} : \forall j \in \mathbb{N} : \sigma_j = (ac, i) \Rightarrow \exists k > j : \sigma_k = (rel, i)$. The automaton $\mathcal{A}_{ga}^{fair}$ correctively$_{\sqsubseteq}^{\mathcal{S}}$ enforces the general availability property.*

Finally, a static analysis may determine that every possible execution of the target program eventually terminates. If this is the case, an even more corrective enforcement paradigm is available to the monitor, which can acquire resources as needed, and close them at the end of the execution. Since every execution if finite, the number ac actions inserted into the sequence will also necessarily be finite.

Let $\mathcal{A}_{ga}^{\Sigma^*} = \langle \Sigma, Q, q_0, \delta_e \rangle$ where Σ is defined as above and

- $Q : \wp(N)$ is the set of resources which have been acquired but not yet released;
- $q_0 = \emptyset$ is the initial state;
- $\delta_{\Sigma^\infty} : Q \times \Sigma \to Q \times \{\Sigma \cup \epsilon\}$ is given as:

$$\delta_e(q,a) = \begin{cases} (q \cup \{i\}, (ac, i); a) & \text{if } a = (use, i) \wedge (ac, i) \notin acts(\tau) \\ (q \cup \{i\}, a) & \text{if } a = (ac, i) \\ (q \backslash i, a) & \text{if } a = (rel, i) \\ (q, a) & \text{if } a = (use, i) \wedge i \in q \\ (\epsilon, f(q)) & \text{if } a = a_{end} \end{cases}$$

Where $f : \wp(\mathcal{I}) \to \Sigma^*$ is a function returning a sequence of the form $rel_i, rel_j, rel_k ...$ for all resources present in its input.

Proposition 8. *The automaton $\mathcal{A}_{ga}^{\Sigma^*}$ correctively$_{\sqsubseteq}^{\Sigma^*}$ enforces the general availability property.*

6 Conclusion and Future Work

In this paper, we propose a framework to analyze the security properties enforceable by monitors capable of transforming their input. By imposing constraints on the enforcement mechanism to the effect that some behaviors existing in the input sequence

must still be present in the output, we are able to model the desired behavior of real-life monitors in a more realistic and effective way. We also show that real life properties are enforceable in this paradigm, and give four examples of relevant real-life properties.

The framework presented in this paper allows us to transform a program execution to ensure its compliance with a security policy, while reasonably approximating the semantics of the execution. We believe this framework to be sufficiently flexible to be useful in other program rewriting contexts, and even in situations where security is not the main concern, such as controller synthesis or specification refinement. In future work, we hope to adapt our corrective framework to such contexts.

References

1. L. Bauer, J. Ligatti, and D. Walker. More enforceable security policies. In *proceedings of the Foundations of Computer Security Workshop*, July 2002.
2. D. Beauquier, J. Cohen, and R. Lanotte. Security policies enforcement using finite edit automata. *Electronic Notes in Theoretical Computer Science*, 229(3):19–35, 2009.
3. N. Bielova, F. Massacci, and A. Micheletti. Towards practical enforcement theories. In *Proceedings of the 14th Nordic Conference on Secure IT Systems, NordSec 2009*, volume 5838 of *LNCS*, pages 239–254. Springer, October 2009.
4. W. E. Boebert and R. Y. Kain. A practical alternative to hierarchical integrity policies. In *Proceedings of the 8th National Computer Security Conference*, September 1985.
5. D. F. C. Brewer and M. J. Nash. The chinese wall security policy. In *IEEE Symposium on Security and Privacy*, pages 206–214, 1989.
6. H. Chabot, R. Khoury, and N. Tawbi. Generating in-line monitors for Rabin automata. In *Proceedings of the 14th Nordic Conference on Secure IT Systems, NordSec 2009*, volume 5838 of *LNCS*, pages 287–301. Springer, October 2009.
7. P. Fong. Access control by tracking shallow execution history. In *proceedings of the 2004 IEEE Symposium on Security and Privacy*, May 2004.
8. K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *ACM Transactions on Programming Languages and Systems*, 28(1):175–205, 2006.
9. R. Khoury and 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.
10. J. Ligatti, L. Bauer, and D. Walker. Enforcing non-safety security policies with program monitors. In *proceedings of the 10th European Symposium on Research in Computer Security*, September 2005.
11. J. Ligatti and S. Reddy. A theory of runtime enforcement, with results. Technical Report USF-CSE-SS-102809, University of South Florida, apr 2010.
12. F. B. Schneider. Enforceable security policies. *Information and System Security*, 3(1):30–50, 2000.
13. A. E. K. Sobel and J. Alves-Foss. A trace-based model of the chinese wall security policy. In *In Proceedings of the 22nd National Information Systems Security Conference*, 1999.
14. C. Talhi, N. Tawbi, and M. Debbabi. Execution monitoring enforcement under memory-limitations constraints. *Information and Computation*, 206(1):158–184, 2008.