Execution Monitoring Enforcement For Limited-Memory Systems

Chamseddine Talhi
LSFM Group, Computer Science Department
Laval University
(Qc) Quebec, Canada, G1K 7P4
talhi@ift.ulaval.ca

Nadia Tawbi
LSFM Group, Computer Science Department
Laval University
(Qc) Quebec, Canada, G1K 7P4
Nadia.Tawbi@ift.ulaval.ca

Mourad Debbabi
Computer Security Labortory
Concordia Institute for Information Systems Engineering
Concordia University, Montral, (Qc), Canada
debbabi@ciise.concordia.ca

Abstract

Recently, attention has been given to formally characterize security policies that are enforceable by different kinds of security mechanisms. Since execution monitoring (EM) is a ubiquitous technique for enforcing security policies, this class of enforcement mechanisms has attracted the attention of the majority of authors characterizing security enforcement. A very important research problem is the characterization of security policies that are enforceable by execution monitors constrained by memory limitations. This paper contributes to give more precise answers to this research problem. To represent execution monitors constrained by memory limitations, we introduce a new class of automata that we call Bounded History Automata. Characterizing memory limitations gives rise to a precise taxonomy of security policies enforceable under such constraints.

This work is in the same line as the research work advanced by Schneider [20], Ligatti et.al [1, 13] and Fong [9] on security enforcement. Our main contribution consists in (1) instantiating Fong's abstraction idea to deal with memory-limitations, (2) defining Bounded History Automata by applying our abstraction to both security automata and edit automata [1], and (3) Reasoning about the enforcement power of bounded history automata by investigating the enforcement of locally testable properties; a well studied class of languages that are recognizable by investigating "local" information. Our approach gives rise to a realistic evaluation of the enforcement power of execution monitoring. This evaluation is based on bounding the memory size used by the monitor to save execution history, and identifying the security policies enforceable under such constraint.

keywords: *execution monitoring, security policies, edit automata, bounded history automata, locally-testable properties.*

1 Introduction

Securing software platforms is based on specifying a set of security policies and deploying the appropriate mechanisms to enforce them. Enforcement mechanisms can be classified into three main classes; static enforcement, execution monitoring and rewriting [12]. Execution monitors are enforcement mechanisms that operate alongside the execution of untrusted programs, they intercept security relevant events, and intervene when an execution is attempting to violate the policy being enforced. While halting the execution represents the common intervention action to respond to a violation, execution monitors can have the power of inserting actions on behalf of the program or suppressing potentially dangerous actions [3].

The efforts of some pioneer authors [20][12] [2] [9] contribute to the emergence of a new research field that targets characterizing enforcement mechanisms and identifying the classes of enforceable security policies. Since execution monitoring (EM) enjoys the property of being a ubiquitous technique for security policies enforcement, this class of enforcement mechanisms has attracted the attention of the majority of researchers studying formal characterization of security policies and enforcement mechanisms.

A very important research problem is the characterization of security policies that are enforceable by execution monitors constrained by memory limitations. Providing

precise answers to this problem is more crucial when we design a security architecture for embedded systems. The migration of the Java security model to embedded platforms is a concrete witness of the importance of this problem. Indeed, in the absence of a precise evaluation of Java EM-enforcement mechanisms, Java stack inspection mechanism is judged to be too heavy for embedded platforms and was replaced by a lightweight mechanism having less enforcement power [23].

This research problem was not well addressed in the literature since the majority of models make no constraint on the size of the tracked execution history. Fong [9] is the first one who presented an interesting attempt to answer this research problem. He presented a general theoretical framework to characterize security policies that are enforceable by execution monitors constrained by the available information about the execution history. However, the results of Fong are too general and concern only prefix-closed properties over finite executions. In this paper, we present a precise characterization of security policies that are enforceable by monitors constrained by memory limitations. These constraints are represented by limiting the space used by monitors to save the execution history. Our approach allows the characterization of any property over finite or infinite executions that is enforceable by any conventional monitor (that can only halt the execution in response to a potential violation) or any more powerful monitor (that can insert actions on behalf of the program or suppress potentially dangerous actions).

1.1 Related Work

Schneider [20] is the pioneer in characterizing security policies enforceable by execution monitoring. His contribution is mainly twofold: (1) characterizing EM-enforceable policies by security automata, and (2) identifying EM-enforceable policies as a subset of safety properties. Jay Ligatti, Lujo Bauer, and David Walker [1, 13, 14] have introduced *edit automata*; a more detailed framework for reasoning about execution monitoring mechanisms. While Schneider views execution monitors as sequence recognizers, Ligatti et al. view them as sequence transformers. By having the power of modifying program actions at run time, edit automata are provably more powerful than security automata in enforcing security policies [14].

Hamlen et al. [12] provided a taxonomy of enforceable security policies. In this taxonomy, they investigated a larger set of enforcement mechanisms, including static enforcement, execution monitoring and program rewriting. One important contribution of this work is the connection of this taxonomy to the arithmetic hierarchy of computational complexity theory. A second important contribution of this work is a more accurate characterization of security policies that are actually enforceable by execution monitoring.

Fong [9] is the first one who provided a fine-grained, information-based characterization of EM-enforceable policies. To represent constraints imposed on information available to execution monitors, he used abstraction functions over sequences of controlled programs. To compare classes of EM-enforceable security policies, he defined a lattice on the space of all congruence relations over actions sequences. However, the EM-enforceable policies investigated by Fong are limited to safety properties over finite executions. Although this work provided a nice and elegant information-based classification of EM-enforceable policies, real constraints on information tracked by execution monitors are not investigated. Indeed, the only abstraction notion investigated by Fong is the mapping of action sequences into actions sets.

1.2 Contributions

In this paper, we propose a characterization of security policies that are enforceable by execution monitors constrained by memory limitations. Namely, we characterize such memory limitation constraints by the memory size available to save the execution history. Our main contributions are the following:

- We instantiate Fong's abstraction idea to deal with memory-limitations. Namely, we characterize the information tracked by an execution monitor by a bounded history representing a limited space used by the monitor to store the execution history.
- We apply our abstraction to security automata [20] and edit automata [1] [14]. The result is a new class of automata that we call *bounded history automata* including two subclasses; *bounded security automata* and *bounded edit automata*.
- We identify a new taxonomy of EM-enforceable properties that is directed by the memory size used by execution monitors to save execution history. This taxonomy gives rise to a realistic evaluation of the execution monitoring enforcement power. This evaluation is based on bounding the space size used by the monitor to save execution history, and identifying the security policies enforceable under such constraint.
- We demonstrate the power of bounded edit automata enforcement by investigating locally testable properties [4, 8]; a well studied class of properties that are recognizable by inspecting "local" information of bounded size. Defining the connection between *BEA* enforceable properties and locally testable properties allows us to benefit from a panoply of results available in language theory and concerning locally testable

properties recognition. Especially, we can benefit from the algorithms deciding whether a property is locally testable and those identifying the locality order of a given locally testable property (i.e. the bounded size of local information needed to recognize the property sequences) [21] [22] [24].

The remainder of this paper is organized as follows. In section 2, we present the main definitions needed alongside the paper. Section 3 is dedicated to the presentation of the main characterizations of execution monitoring enforcement. Section 4 is devoted to the presentation of bounded history automata. In section 5, we investigate EM-enforcement of locally testable properties. We present some examples of *BHA*-enforceable policies in section 6 and we end by the conclusion and the future work in section 7.

2 Definitions

We start by some notations of language theory. An alphabet Σ is a set of symbols. Those symbols are used to represent program actions. In the sequel, we use interchangeably symbols and actions. We denote the set of finite sequences over Σ by Σ^*. The set of all infinite sequences over Σ is denoted by Σ^ω, and $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$ denotes the set of all finite and infinite sequences over Σ. The empty sequence is denoted by ϵ. A sequence counting only one input action *a* is denoted by "a". We denote by $\sigma\sigma'$ the concatenation of two sequences σ and σ'. A language L over Σ is a subset of Σ^∞. We denote by $LL' = \{\sigma\sigma' | \sigma \in L \wedge \sigma' \in L'\}$ the concatenation of two languages L and L'. The intersection of two languages L and L' is denoted by $L \cap L' = \{\sigma | \sigma \in L \wedge \sigma \in L'\}$. The union of two languages L and L' is denoted by $L \cup L' = \{\sigma | \sigma \in L \vee \sigma \in L'\}$. The difference of two languages L and L' is denoted by $L \setminus L' = \{\sigma | \sigma \in L \wedge \sigma \notin L'\}$. We denote by $|\sigma|$ the length of a sequence σ. The set $\Sigma^k = \{\sigma \in \Sigma^* : |\sigma| = k\}$ denotes the set of all possible sequences of length *k* where *k* is a positive integer. For some positive integer *k*, $\Sigma_k = \{\sigma \in \Sigma^* : |\sigma| \leq k\}$ denotes the set of all possible sequences of length less than or equal to *k*. The set $(\Sigma_k \times \Sigma_k)_k = \{(\sigma, \sigma') \in \Sigma_k \times \Sigma_k : |\sigma\sigma'| \leq k\}$ denotes the set of all possible pairs of sequences such that the length of the concatenation of the two sequences is less than or equal to *k* where *k* is a positive integer.

A sequence σ' is a *prefix* of another sequence σ if there exists a sequence σ'' such that $\sigma = \sigma'\sigma''$. Similarly, σ' is a *suffix* of σ if there exists a sequence σ'' such that $\sigma = \sigma''\sigma'$. We denote by $\sigma[..k]$ the length-k prefix of σ where *k* is a positive integer. Similarly, $\sigma[k+1..]$ denotes the suffix of σ consisting of all but the first *k* symbols of σ. We denote by $Pref(\sigma)$ the set of all prefixes of a sequence σ. Similarly, $Suf(\sigma)$ denotes the set of all suffixes of a sequence σ. A *k* length factor of σ starting at position *i* is denoted by $\sigma[i..i+k-1]$. For a sequence σ and a positive integer, the set of all factors of length *k* of σ is denoted by $Fact^k(\sigma) = \{\sigma' \in \Sigma^k | \exists\, \sigma'' \in \Sigma^*. \exists \sigma''' \in \Sigma^\infty : \sigma = \sigma''\sigma'\sigma'''\}$. The set $Pref_k(\sigma) = \{\sigma' \in Pref(\sigma) : |\sigma'| \leq k\}$ denotes the set of all prefixes of σ of length less than or equal to *k* where $k \leq |\sigma|$. Similarly, the set of suffixes of σ of length less than or equal to *k* is denoted by $Suf_k(\sigma) = \{\sigma' \in Suf(\sigma) : |\sigma'| \leq k)\}$.

We need also some definitions related to security policies. Let Σ denote the set of all input actions that can be intercepted by a monitor. A security policy $P \subseteq \Sigma^\infty$ is a set of sequences. A sequence σ satisfies a security property *P* if and only if $\sigma \in P$. A security policy *P* is a *property* if there exists a predicate $\hat{P}$ over individual executions satisfying $\forall \sigma \in \Sigma^\infty. \sigma \in P \Leftrightarrow \hat{P}(\sigma)$. A security property *P* is prefix-closed if and only if: $\forall \sigma \in \Sigma^\infty. \sigma \in P \Rightarrow Pref(\sigma) \subseteq P$.

3 EM-Enforcement Characterization

In this section, we present the two main characterizations of EM-enforcement: *security automata* (*SA*) and *edit automata* (*EA*). We present also the Fong's information-based characterization of *SA*-enforceable policies.

3.1 Security automata

In this characterization, a monitor can intervene only by halting the program execution. According to this definition of EM-enforcement, Schneider [20] observes that every EM-enforceable security policy *P* must be a prefix-closed property. An EM-enforceable policy is specified by a *security automaton*.

Definition 3.1 (Security Automaton) *A Security Automaton (SA)[20] is a quadruple $\langle \Sigma, Q, q_0, \delta \rangle$ where:*

- *Σ is the set of finite or countably infinite input actions.*
- *Q is the set of finite or countably infinite automaton states.*
- *$q_0 \in Q$ is the initial state.*
- *$\delta : (Q \times \Sigma) \rightarrow Q$ is the (possibly partial) transition function. For a state q and an input action a, $\delta(q, a)$ is defined if the monitor is supposed to accept the input action a while it is in the state q. The transition $\delta(q, a)$ is not defined if the monitor is supposed to halt after reading the input action a while being in the state q.*

A sequence of input actions is accepted (recognized) by a security automaton if, starting from state q_0 and reading the sequence one input action at a time, a transition is defined for each input action in the sequence and the reached

state. The automaton state changes according to each taken transition. This acceptance definition is broad enough to cover finite and infinite sequences recognition and is represented by a recognition path. A recognition path is a (finite or infinite) sequence of transition steps of the form $q \xrightarrow{a} q'$ where $q' = \delta(q, a)$.

In addition to specifying security policies, security automata can serve as the basis for an execution monitor implementation. In such implementation, before performing any protected action, an input symbol is sent to the simulation of the security automata: If the security automata can make a transition then the controlled program is allowed to perform the protected action. Otherwise, the controlled program is terminated (since it is attempting to violate the security policy).

3.2 Edit Automata

Edit automata characterize execution monitors that in addition to halting the execution of the controlled program, can modify the program actions either by suppressing or inserting actions.

Definition 3.2 (Edit Automaton) *An edit automaton (EA) is defined by a quadruple $\langle \Sigma, Q, q_0, \delta \rangle$ where:*

- *Σ is the set of finite or countably infinite input actions.*
- *Q is the set of finite or countably infinite automaton states.*
- *$q_0 \in Q$ is the initial state.*
- *$\delta : (Q \times \Sigma) \rightarrow (Q \times \Sigma^\infty)$ is the transition function[1]. For a state q and an input action a, $\delta(q, a)$ is defined by:*

$$\delta(q,a) = \begin{cases} (q', \sigma) \text{ where } \sigma \neq a \wedge \sigma \neq \epsilon & \textit{(Insert)} \\ (q', a) & \textit{(Accept)} \\ (q', \epsilon) & \textit{(Suppress)} \\ \textit{Undefined, otherwise.} & (Halt) \end{cases}$$

When given a current state and an input action, the transition function specifies a new state to enter and a sequence of actions to execute. The transition function specifies the intervention action to take in order to enforce the property. The (Accept) case corresponds to accepting the input action, the (Suppress) case corresponds to suppressing the input action, the (Insert) case corresponds to inserting a sequence of actions, and the (Halt) case corresponds to halting the execution. The automaton accepts a sequence σ if it can follow a valid path while reading the input actions of σ. The acceptance path is a (finite or infinite) sequence of transition steps of the form $q \xrightarrow[\tau]{a} q'$ where *a* is the input action and τ is the sequence edited by the automaton. Thus a transition step can represent three cases. (1) If $\tau =$ "a" then the input action *a* is accepted, (2) if $\tau \neq \epsilon \wedge \tau \neq$ "a" then the sequence τ is inserted, and (3) if $\tau = \epsilon$ then the input action *a* is suppressed. We denote by $A(\sigma)$ the sequence edited by an edit automaton *A* while reading an input sequence σ and we denote by A_P the property enforced by *A*. It is important to remind two principles that any monitor must obey in order to ensure *effective* enforcement [1]:

1. *Soundness*: Any observable execution must satisfy the property being enforced.
2. *Transparency*: The semantics of any execution satisfying the property must be preserved.

Since, edit automata can modify input sequences, *EA* ensure *Soundness* by transforming bad executions into valid executions, and ensure *transparency* by transforming valid executions into equivalent valid executions. Let us denote by *effective$_\cong$ enforcement* the effective enforcement of edit automata based on a given equivalence relation $\cong$. Let us recall the formal definition of effective$_\cong$ enforcement [1]:

Definition 3.3 (Effective$_\cong$ Property Enforcement) *Let $A = \langle \Sigma, Q, q_0, \delta \rangle$ be an edit automaton and $\cong$ an equivalence relation over sequences of Σ^∞. The EA effectively$_\cong$ enforces P if and only if, for each sequence $\sigma \in \Sigma^\infty$ we have:*

1. *$A(\sigma) \in P$, and*
2. *$\sigma \in P \Rightarrow A(\sigma) \cong \sigma$.*

It has been proved [1][14] that edit automata are powerful effective$_=$ enforcers. Indeed, an edit automaton can suppress a sequence of potentially dangerous actions until it can confirm that the sequence is legal, at which point it inserts all the suppressed actions [1] [2], [14][2]. A lower bound of properties that are effectively$_=$ enforceable by edit automata is identified and called *renewal properties*.

Definition 3.4 *A property P over Σ^∞ is a renewal property if and only if one of the following two equivalent conditions is satisfied:*

[1]The transition function definition presented here is equivalent to the original definition of Ligatti et all [1]. The only difference is that an input action in our definition is consumed at each transition step while it is not consumed in an insert step of their definition. However, for any edit automaton based on the definition in [1], it is easy to construct an equivalent automaton according to our definition. We adopt the definition presented here mainly for three reasons: (1) to be closer to automata theory and (2) to facilitate automata construction in the proofs presented in this paper and (3) to present a definition that is more suited to effective$_=$ enforcement (3.3) than the original definition.

[2]Even if effective$_=$ enforcement has been formally defined later in [13], the proof presented in [1] uses implicitly effective$_=$ enforcement.

$\forall \sigma \in \Sigma^{\omega}.\ \sigma \in P \Leftrightarrow Pref(\sigma) \cap P$ *is an infinite set (RW$_1$)*

$\forall \sigma \in \Sigma^{\omega}.\sigma \in P \Leftrightarrow \forall \sigma' \in Pref(\sigma).$
$\exists \sigma'' \in Pref(\sigma).\sigma' \in Pref(\sigma'') \wedge \sigma'' \in P$ *(RW$_2$)*

According to this definition, any property over finite executions is a renewal property.

Proposition 3.5 *[14] A property P over Σ^{∞} is effectively$_{=}$ enforceable by edit automata if P is a renewal property and $\epsilon \in P$.*[3]

In the rest of this paper, any mention of *enforcement* refers to *effective$_{=}$ enforcement*.

3.3 Fong's Characterization

Fong [9] has proposed an information-based approach characterizing EM-enforceable security polices by the information consumed by execution monitors. He first introduced Shallow history automata (*SHA*); a new security automata class characterizing security policies that are enforceable by monitors tracking shallow access history. The information provided by a shallow access history is whether an action has been previously granted. The formal definition of shallow history automata is the following.

Definition 3.6 *[9] A shallow history automaton is a security automaton of the form $\langle \Sigma, 2^{\Sigma}, \emptyset, \delta \rangle$ where:*

- *Σ is the set of finite or countably infinite input symbols.*
- *2^{Σ} is the set of finite or countably infinite automaton states. Each state represents a shallow history.*
- *$\emptyset$ is the initial (shallow history) state.*
- *$\delta : 2^{\Sigma} \times \Sigma \rightarrow 2^{\Sigma}$ is the transition function. δ is defined such that: $\forall\ H \in 2^{\Sigma}.\forall a \in \Sigma.\ \delta(H, a) =$*
  $\begin{cases} H \cup \{a\} \text{ if } H \cup \{a\} \text{ is a valid shallow history} & (1) \\ \text{Undefined, otherwise.} & (2) \end{cases}$

Motivated by the enforcement power of *SHA*, Fong generalized the technique to provide an information-based classification of EM-enforceable policies. The classification criteria are the constraints on the information that is available to one execution monitor. To represent the information available to an execution monitor, a set of abstract states is used. An abstraction function α is defined so that each abstract state represents a set of different finite executions. By matching different action sequences onto a single action sequence, the abstraction function reduced the set of action sequences that are visible to an execution monitor, hence resulting in a reduction of the set of security policies enforceable by this execution monitor. An abstraction function can be defined by the following:

[3] Proposition 3.5 corresponds to theorem 8 in [14].

Definition 3.7 (Abstraction Function) *[9]*

Let S be a finite or countably infinite set of abstract states and let α be any function such that $\alpha : \Sigma^* \rightarrow S$. α is an abstraction function if it satisfies the following compatibility property:

$$\forall w, w' \in \Sigma^*.\forall a \in \Sigma.\alpha(w) = \alpha(w') \Rightarrow \alpha(wa) = \alpha(w'a)$$

The security automaton specifying the behavior of an execution monitor tracking the abstract states is defined by an α*-SA*.

Definition 3.8 (α-SA) *[9]*

Let $\alpha : \Sigma^* \rightarrow S$ be a compatible abstraction function. An α-SA is a SA $\langle \Sigma, S, \alpha(\epsilon), \delta \rangle$ such that for all $w \in \Sigma^*$ and for all $a \in \Sigma$, either $\delta(\alpha(w), a) = \alpha(wa)$ or $\delta(\alpha(w), a)$ is not defined at all.

The α*-SA*-enforceable security policies are those that can be enforced by monitors consuming information left behind by the abstraction function[9].

4 Bounded History Automata

In this section, we present Bounded History Automata (*BHA*). (*BHA*) is a class of automata characterizing security policies that are enforceable by monitors manipulating bounded space to track execution histories. Within this class, we identify two main classes: Bounded Security Automata (*BSA*) and Bounded Edit Automata (*BEA*). To characterize a monitor tracking bounded histories of length *k*, the *BHA* states set and the transition function are defined such that:

- Each state represents a bounded history of a (possibly infinite) set of valid executions.
- For each bounded history *h* and each input action *a*, the new history *h'* (if it is defined for *h* and *a*) defined by the transition function is an abstraction of the history *ha*.

4.1 Bounded Security Automata

Definition 4.1 (Bounded Security Automaton) *A BSA of bound k (k-BSA) is a SA $\langle \Sigma, Q = \Sigma_k, q_0, \delta \rangle$ where:*

- *Σ is the set of finite or countably infinite input actions.*
- *Q is the set of finite or countably infinite automaton states. Each state in Q represents a bounded history of a (possibly infinite) set of accepted sequences.*
- *k defines the maximum size of a history.*
- *q_0 is the initial state (usually the empty history ϵ).*

- $\delta : (Q \times \Sigma) \rightarrow Q$ *is the (possibly partial) transition function.*

Intuitively, when a *BSA A*, is in the state *h* and reading an input action *a*, if a transition is defined from *h* to a state *h'* such that $h' = \delta(h, a)$ then *h'* is an abstraction of the history *ha*. The meaning of this abstraction is that only the abstraction *h'* of *ha* is relevant for the enforcement of the security policy A_P in any extension of *ha* where any abstraction must be formatted as a sequence of at most *k* actions. Thus, the transition function δ defines an abstraction function $\beta : \Sigma_{k+1} \rightarrow \Sigma_k$ where $\delta(h, a) = \beta(ha)$. We denote the abstraction function defined by the transition function of a *BHA A* by A_β.

We denote by A_P the security policy enforced by a *BSA A*. Thus, A_P is the set of all (finite or infinite) sequences accepted by *A*. If we consider only finite sequences, we denote by $A_{P_f} \subseteq A_P$ the set of all finite sequences of A_P. Since we are dealing with a class of security automata, the set of security properties enforceable by *BSA* is a subset of the safety properties set. Let EM_{kSA} denote the set of properties enforceable by bounded security automata of bound *k*.

Proposition 4.2 *For any two positive integers k and k' such that $k < k'$, we have $EM_{kSA} \subset EM_{k'SA}$.*

In what follows, we explain the connection between *BSA* and Fong's α-*SA* and *SHA*. The definitions of abstraction functions, α-*SA*, and *SHA* have been presented in 3.7, 3.8, and 3.6 respectively.

Proposition 4.3 *For any BSA $A = \langle \Sigma, Q = \Sigma_k, q_0, \delta \rangle$, there exist an α-SA enforcing A_{P_f}.*

Proposition 4.4 *For any α-SA $A = \langle \Sigma, Q, \alpha(\epsilon), \delta \} \rangle$ enforcing a property P, there exists a k-BSA enforcing the same property P where k is the maximum size that can have the encoding of an abstract state.*[4]

Proposition 4.5 *If the set of input actions Σ is finite such that $|\Sigma| = k$, then, for any shallow history automaton enforcing a property P, there exists a k-BSA enforcing P.*

Proof: We prove this result by constructing the *BSA* enforcing *P*. Let $A = \langle \Sigma, 2^\Sigma, \emptyset, \delta \rangle$ be the shallow history automaton enforcing the property *P*. The *BSA* enforcing *P* is defined by $\langle \Sigma, \Sigma_k, \epsilon, \delta' \rangle$ where the transition function δ' is defined such that: $\forall \sigma \in \Sigma_k . \forall a \in \Sigma .\ \delta'(\sigma, a) =$

$$\begin{cases} \sigma \text{ if } a \in \mathit{Act}(\sigma) & (1) \\ \sigma a \text{ if } a \notin \mathit{Act}(\sigma) \wedge \delta(\mathit{Act}(\sigma), a) = \mathit{Act}(\sigma) \cup \{a\} & (2) \\ \text{Undefined, otherwise.} & (3) \end{cases}$$

[4]For space-limitation, we do not present the proofs of the majority of the propositions presented in this paper.

where $\mathit{Act} : \Sigma_k \rightarrow 2^\Sigma$ is the function that returns for each sequence σ the set of actions present in σ.

■

For finite input actions sets, *BSA* have more enforcement power than *SHA*. Indeed, any *BSA*-enforceale property distinguishing between two sequences counting the same set of actions, is not enforceable by any *SHA*.

4.2 Bounded Edit Automata

As we did for *BSA*, we use the automaton states to represent bounded histories. The only difference here is that a bounded history is a concatenation of two sequences; the first is accepted by the automaton, and the second is suppressed in order to reinsert it if a valid prefix is recognized. To define a bounded edit automaton *BEA*, we use the construction technique adopted in [14].

Definition 4.6 *A BEA of bound k (k-BEA) is an edit automaton $\langle \Sigma, Q = (\Sigma_k \times \Sigma_k)_k, q_0, \delta \rangle$ where:*

- Σ *is the set of finite or countably infinite input actions.*
- Q *is the set of finite or countably infinite automaton states. Each state is a pair $\langle \sigma_{Acc}, \sigma_{Sup} \rangle$ where σ_{Acc} represents a suffix of a (possibly infinite) set of accepted prefixes and σ_{Sup} represents a suppressed suffix. Since we are dealing with bounded histories, $\sigma_{Acc}\sigma_{Sup} \in \Sigma_k$.*
- k *defines the maximum size of a history.*
- $q_0 \in Q$ *is the initial state, usually the pair $\langle \epsilon, \epsilon \rangle$ which means that no prefix was accepted and no sequence was suppressed.*
- $\delta : (Q \times \Sigma) \rightarrow Q$ *is the (possibly partial) transition function.*

For a state $\langle \sigma_{Acc}, \sigma_{Sup} \rangle$ and an input action *a*, the new state $\langle \sigma'_{Acc}, \sigma'_{Sup} \rangle$ is defined by $\delta(\langle \sigma_{Acc}, \sigma_{Sup} \rangle, a)$ such that the history $\sigma'_{Acc}\sigma'_{Sup}$ is an abstraction of $\sigma_{Acc}\sigma_{Sup}a$. We denote by $\beta : \Sigma_{k+1} \rightarrow \Sigma_k$ the abstraction function used to define δ such that $\delta(\langle \sigma_{Acc}, \sigma_{Sup} \rangle, a) = \alpha(\beta(\sigma_{Acc}\sigma_{Sup}a))$ where the function $\alpha : \Sigma_k \rightarrow (\Sigma_k \times \Sigma_k)_k$ specifies the accepted sequence and the suppressed one depending on the property being enforced by the *BEA*. Let EM_{kEA} denote the set of properties enforceable by bounded edit automata of bound *k*.

Proposition 4.7 *For any two positive integers k and k' such that $k < k'$, we have $EM_{kEA} \subset EM_{k'EA}$.*

4.3 Bounded-History-Based Taxonomy of EM-Enforceable Policies

Proposition 4.2 and proposition 4.7 together, allow us to identify a new taxonomy of EM-enforceable policies that is based on memory limitation constraints. Indeed, if we denote by EM_{SA} the class of properties that are enforceable by *SA*, then by proposition 4.2, we get the following taxonomy: $EM_{0SA} \subset EM_{1SA} \subset EM_{2SA} \ldots \subset EM_{SA}$. The smallest class of this taxonomy is the class of properties that are enforceable by *BSA* having no space to save the execution history and the biggest class is the class of properties that are enforceable by security automata which have no constraint on the space used to save the execution history. Similarly, if we denote by EM_{EA} the class of properties that are enforceable by *EA*, then by proposition 4.7, we get the following taxonomy: $EM_{0EA} \subset EM_{1EA} \subset EM_{2EA} \ldots \subset EM_{EA}$. Note that for any positive integer *k*, we have $EM_{kSA} \subset EM_{kEA}$.

5 Bounded History Automata and Local Testability

In language theory, *locally testable* properties [4] are identified as the class of properties that are recognizable by inspecting "local" information. In the following, we investigate the connection between locally testable properties and *BHA*-enforceable properties.

5.1 Locally Testable Properties

Locally testable properties *LT* are properties that are recognizable by *scanners*; automata equipped with a finite memory and a *sliding* window of a fixed length *n* [4]. To analyze a sequence, the sliding window is moved from left to right on the input sequence. During the analysis of an input sequence, the scanner remembers the prefixes or suffixes of length smaller than *n* and the factors of length *n*. Depending on the identified sets of prefixes, suffixes, and factors, the scanner decides to accept or to reject the input sequence. In the sequel, we present the definition of locally testable properties and the different classes of LT properties that we can found in the literature.

Definition 5.1 (Locally Testable Properties) [5]

Let k be a positive integer. A property L of Σ^∞ is k-locally testable if there exist four sets $P, S \subseteq \Sigma^{k-1}$, $X \subseteq \Sigma_{k-1}$ and $F \subseteq \Sigma^k$ such that the elements of L are defined by the two following rules:

- $\forall \sigma \in \Sigma^*.\ \sigma \in L \setminus \{\epsilon\} \Leftrightarrow$
  $(\sigma \in X) \vee ((Pref_{k-1}(\sigma) \cap P \neq \emptyset) \wedge$
  $(Suf_{k-1}(\sigma) \cap S \neq \emptyset) \wedge (Fact^k(\sigma) \subseteq F))$.
- $\forall \sigma \in \Sigma^\omega.\ \sigma \in L \setminus \{\epsilon\} \Leftrightarrow \forall \sigma' \in Pref(\sigma).$
  $\exists \sigma'' \in Pref(\sigma).\sigma' \in Pref(\sigma'') \wedge \sigma'' \in L$.

A property L of Σ^∞ is locally testable if it is k-locally testable for some integer k.

According to this definition, the set of non empty finite sequences of a locally-testable property *L* is defined by the set $(L \cap \Sigma^*) \setminus \{\epsilon\} = ((P\Sigma^* \cap \Sigma^* S) \setminus \Sigma^* \overline{F} \Sigma^*) \cup X$ where $\overline{F} = \Sigma^k \setminus F$. Similarly, the set of infinite sequences of *L* is defined by the set $L \cap \Sigma^\omega = (P\Sigma^\omega \cap (\Sigma^* S)^\omega) \setminus \Sigma^* \overline{F} \Sigma^\omega$.

We suppose that $X = \{\sigma \in L : |\sigma| < k\}$ i.e. all sequences of *L* of length less than *k* are in *X*. Intuitively, a sequence σ is in *L* if it satisfies one of the two following conditions depending on its length:

- Case $|\sigma| < k$: σ must be an element of *X* ($\sigma \in X$).
- Case $|\sigma| \geq k$: σ must satisfy the three following conditions:
  - ($Pref_k(w) \cap P \neq \emptyset$), i.e. σ must have one of its prefixes in *P*.
  - ($Suf_k(w) \cap S \neq \emptyset$), i.e. must have one of its suffixes in *S* .
  - ($Fact^k(w) \subseteq F$), i.e. σ must have all its factors of length *k* in *F* .

The following proposition is important for the recognition of *LT* properties:

Proposition 5.2 *Let $L = ((P\Sigma^* \cap \Sigma^* S) \setminus \Sigma^* F \Sigma^*) \cup X$ be a k-locally testable property. Let us define the two sets $F_{initial}$ and $F_{terminal}$ by:*

- $F_{initial} = \begin{cases} \{f \in F \mid Pref(f) \cap P \neq \emptyset\} & \text{if } P \neq \emptyset \\ F & \text{if } P = \emptyset \end{cases}$
- $F_{terminal} = \begin{cases} \{f \in F \mid Suf(f) \cap S \neq \emptyset\} & \text{if } P \neq \emptyset \\ F & \text{if } S = \emptyset \end{cases}$

A sequence σ such that $|\sigma| = k + d$ for some positive integer d, is an element of L if and only if:

(1) $\exists f_{init} \in F_{initial} \wedge \exists \sigma' \in \Sigma^* : \sigma = f_{init}\sigma'$, *and*

(2) $\exists f_{end} \in F_{terminal} \wedge \exists \sigma' \in \Sigma^* : \sigma = \sigma' f_{end}$, *and*

(3) $\forall 1 \leq i \leq d+1 : \sigma[i..i+k-1] \in F$.

[5] We present here a definition that covers both finite and infinite sequences. In the literature, locally testable properties over finite sequences and locally testable properties over infinite sequences are treated separately [8] [17].

$F_{initial}$ represents the set of all factors of length *k* that can be accepted as prefixes of a sequence of *L*. $F_{terminal}$ represents the set of all factors of length *k* that can be accepted as suffixes of a sequence of *L*. Condition (1) ensures that any sequence σ of *L* such that $|\sigma| \geq k$ must start by a prefix of *P* (a factor of $F_{initial}$). Condition (2) ensures that σ must end by a suffix of *S* (a factor of $F_{terminal}$), and (3) ensures that all factors of length *k* of σ must be elements of F. The property $L = (((\{ab\}\Sigma^* \cap \Sigma^*\{ba\}) \setminus \Sigma^*\{aaa, abb, bab, bba, bbb\}\Sigma^*) \cup \{aa, bb\}$ is an example of a *LT* property where $P = \{ab\}$, $S = \{ba\}$, $X = \{aa, bb\}$, $F = \{aba, baa, aab\}$, $\overline{F} = \{aaa, abb, bab, bba, bbb\}$, and $F_{initial} = F_{terminal} = \{aba\}$. The property L can also be written as $L = \{aba\}^* \cup \{aa, bb\}$.

Since the definition of a locally testable property L makes no constraints on the sets P, X, S and F, some factors of F cannot be factors of any sequence of L. This can happen when a factor f is accepted as a factor of any sequence of the language, but no sequence starting by a prefix of P and having all its factors in F can have f as a factor. for example, if for the property LL defined above, the set F is defined such that $F = \{aba, baa, aab, bbb\}$, then there is no sequence σ of LL counting bbb as a factor.

For some results of this section we need the exact definition of the factors that are really used to construct a locally testable property. Let L be a k-locally testable property defined by the sets P, S, F, and X. We define the set of factors that are really used to construct the sequences of L by the set F_R defined by:

$$F_R = \{f \in F | \exists \sigma \in \Sigma^*.(\sigma f \in L \vee \exists \sigma' \in \Sigma^+.\sigma f \sigma' \in L).$$

Among *LT* properties, we can identify four main classes: *prefix testable*, *suffix testable*, *prefix-suffix testable*, and *strongly locally testable* properties [17]. We start by *prefix testable* properties that are recognizable by inspecting only prefixes of limited size.

Definition 5.3 (Prefix Testable Properties) *Let k be a positive integer. A property L of Σ^∞ is k-prefix testable if there exist two sets $P \subseteq \Sigma^k$ and $X \subseteq \Sigma_{k-1}$ such that $L \setminus \{\epsilon\} = P\Sigma^\infty \cup X$ where the elements of L are defined by the following rule:*

$$\forall \sigma \in \Sigma^\infty.\, \sigma \in L \setminus \{\epsilon\} \Leftrightarrow \sigma \in X \vee Pref_k(\sigma) \cap P \neq \emptyset.$$

A property L of Σ^* is prefix locally testable if it is k-prefix testable for some integer k.

According to this definition, the set of non empty finite sequences of a prefix testable property *L* is defined by the set $(L \cap \Sigma^*) \setminus \{\epsilon\} = P\Sigma^* \cup X$. Similarly, the set of infinite sequences of L is defined by the set $L \cap \Sigma^\omega = P\Sigma^\omega$. Respectively, *suffix testable* properties are recognizable by inspecting suffixes of limited size.

Definition 5.4 (Suffix Testable Properties) *Let k be a positive integer. A property L of Σ^∞ is k-suffix testable if there exist two sets $S \subseteq \Sigma^k$ and $X \subseteq \Sigma_{k-1}$ such that the elements of L are defined by:*

- $\forall \sigma \in \Sigma^*.\, \sigma \in L \setminus \{\epsilon\} \Leftrightarrow \sigma \in X \vee Suf_k(\sigma) \cap S \neq \emptyset.$
- $\forall \sigma \in \Sigma^\omega.\, \sigma \in L \setminus \{\epsilon\} \Leftrightarrow \forall \sigma' \in Pref(\sigma).$ $\exists \sigma'' \in Pref(\sigma).\, \sigma' \in Pref(\sigma'') \wedge \sigma'' \in L.$

A property L of Σ^* is suffix testable if it is k-suffix testable for some integer k.

According to this definition, the set of non empty finite sequences of a suffix testable property *L* is defined by the set $(L \cap \Sigma^*) \setminus \{\epsilon\} = \Sigma^* S \cup X$. Similarly, the set of infinite sequences of L is defined by the set $L \cap \Sigma^\omega = (\Sigma^* S)^\omega$. By inspecting both prefixes and suffixes of limited size, we have the class of *prefix-suffix* testable properties:

Definition 5.5 (Prefix-Suffix Testable Properties) *Let k be a positive integer. A property L of Σ^∞ is k-prefix-suffix testable if there exist three sets $P, S \subseteq \Sigma^k$ and $X \subseteq \Sigma_{k-1}$ such that the elements of L are defined by:*

- $\forall \sigma \in \Sigma^*.\, \sigma \in L \setminus \{\epsilon\} \Leftrightarrow \sigma \in X \vee (Pref_k(\sigma) \cap P \neq \emptyset \wedge Suf_k(\sigma) \cap S \neq \emptyset).$
- $\forall \sigma \in \Sigma^\omega.\, \sigma \in L \setminus \{\epsilon\} \Leftrightarrow \forall \sigma' \in Pref(\sigma).\, \exists \sigma'' \in Pref(\sigma).\sigma' \in Pref(\sigma'') \wedge \sigma'' \in L.$

A property L of Σ^* is prefix-suffix testable if it is k-prefix-suffix testable for some integer k.

According to this definition, the set of non empty finite sequences of a prefix-suffix testable property *L* is defined by the set $(L \cap \Sigma^*) \setminus \{\epsilon\} = P\Sigma^* S \cup X$. Similarly, the set of infinite sequences of *L* is defined by the set $L \cap \Sigma^\omega = P\Sigma^\omega \cap (\Sigma^* S)^\omega$. The *strongly locally* testable properties are a variety of *LT* properties that are recognizable by inspecting factors of fixed size.

Definition 5.6 (Strongly Locally Testable Properties) *Let k be a positive integer. A property L of Σ^∞ is k-strongly locally testable if there exist a set $F \subseteq \Sigma^k$ such that $\forall \sigma \in \Sigma^\infty.\, \forall \sigma' \in Pref(\sigma).\, Fact^k(\sigma') \subseteq F = \emptyset$.*

A property L of Σ^* is strongly locally testable if it is k-strongly testable for some integer k.

According to this definition, the set of non empty sequences of a strongly locally testable property *L* is defined by the set $(L \cap \Sigma^\infty) \setminus \{\epsilon\} = \Sigma^\infty \setminus \Sigma^* \overline{F} \Sigma^\infty$.

5.2 *BSA*-Enforceable Local Properties

In what follows, we investigate local properties that are enforceable by *BSA*. Since *BSA* is a class of security automata, a local property has to be prefix-closed in order to be *BSA*-enforceable.

Proposition 5.7 (Prefix-closed Local Properties) *Let k be any positive integer, and let $F \subseteq \Sigma^k$, $P, S \subseteq \Sigma^{k-1}$, and $X \subseteq \Sigma_{k-1}$ be the sets used to define a local property L where $X = \{\sigma \in L : |\sigma| < k\}$:*

1. *If L is a k-prefix-testable property defined by the two sets P and X, then L is prefix-closed if and only if $X \cup P$ is prefix-closed.*
2. *If L is a k-locally testable property defined by the sets P, S, X, and F, then L is prefix-closed if and only if:*
   (I) $X \cup F_{initial}$ is prefix-closed, and
   (II) $F_R \subseteq F_{terminal}$.
3. *If L is a k-strongly-locally testable property defined by the set F, then L is prefix-closed.*
4. *If L is a k-suffix testable property defined by the two sets S and X or a k-prefix-suffix testable property defined by the sets P, S and X, then L is not prefix-closed.*

The following propositions identify *BSA*-enforceable *LT* properties and *LT* properties that are not *BSA*-enforceable.

Proposition 5.8 *Any prefix-closed k-prefix testable property L over Σ^∞ that is defined by two sets $P \subseteq \Sigma^k$ and $X \subseteq \Sigma_{k-1}$ where k is any positive integer, is enforceable by some k-BSA.*

Proposition 5.9 *Any k-strongly locally testable property L over Σ^∞ and defined by some set $F \subseteq \Sigma^k$ is enforceable by some k-BSA.*

Proposition 5.10 *Any prefix-closed k-locally testable property L that is defined by the sets $F \subseteq \Sigma^k$, $P, S \subseteq \Sigma^{k-1}$, and $X \subseteq \Sigma_{k-1}$, where k is any positive integer, is enforceable by some k-BSA.*

Proposition 5.11 *Suffix testable properties and prefix-suffix testable properties are not enforceable by BSA.*

5.3 *BEA*-Enforceable Local Properties

In the sequel, we investigate *BEA*-enforceable local properties.

Proposition 5.12 *Any k-prefix testable property where k is any positive integer is enforceable by some k-BEA.*

Proposition 5.13 *Let k be any positive integer and let $F \subseteq \Sigma^k$ be the set used to define a k-strongly locally testable property L over Σ^∞. Then any such k-strongly locally testable property is enforceable by some BEA.*

Proposition 5.14 *Let k be any positive integer and let the four sets $X \subseteq \Sigma_{\leq k-1}$, $P, S \subseteq \Sigma_{k-1}$ and $F \subseteq \Sigma^k$ be the sets used to define a k-locally testable property L over Σ^∞. Then there exists a BEA enforcing L.*

Proposition 5.15 *Suffix testable properties properties are not enforceable by BEA.*

Proposition 5.16 *Prefix-suffix testable properties are not enforceable by BEA.*

The following two propositions show that suffix testable properties and prefix-suffix testable properties can be enforced by "non bounded" edit automata.

Proposition 5.17 *Let k be any positive integer and let $S \subseteq \Sigma_k$ and $X \subseteq \Sigma_{\leq k-1}$ be the sets used to define a k-suffix testable property L over Σ^∞. Then there exists an edit automaton enforcing L.*

Proposition 5.18 *Let k be any positive integer and let $P, S \subseteq \Sigma_k$ and $X \subseteq \Sigma_{\leq k-1}$ be the sets used to define a k-prefix-suffix testable property L over Σ^∞. Then there exists an edit automaton enforcing L.*

5.4 Local EM-Enforceable Properties

In 5.2 and 5.3, we have demonstrated a strong connection between BHA-enforceable properties and local properties. According to the gotten results, one can take any *BHA*-enforceable local property, and construct the *BHA* enforcing it. In order to get the maximum benefit from those results, it is important to investigate EM-enforceable properties that are local properties. Deciding whether a property is locally testable has been well investigated and many deciding algorithms where proposed [24] [22] [21] [15]. Since, those algorithms are usually defined for properties that are expressed in terms of (conventional) automata, we investigate in the sequel the translation of security automata and edit automata into (conventional) automata. Since we are dealing with both finite and infinite sequences, Büchi automata seems to be the automata model that is the most suitable to characterize security policies that are *EA* effectively$_=$-enforceable. We consider only deterministic Büchi automata since all the automata used in this paper [6] are deterministic automata. First we recall the formal definition

[6] Security Automata, edit automata, and bounded history automata

of into Büchi automata and explain their property recognition mode.

Definition 5.19 *[18] [19] [Büchi Automata] A Büchi Automaton is a 5-tuple $\langle \Sigma, Q, I, F, \delta \rangle$ where:*

- Σ *is the set of finite or countably infinite input actions.*
- Q *is the set of finite or countably infinite automaton states.*
- q_0 *is the initial state.*
- $F \subseteq Q$ *is the set of final states.*
- $\delta : (Q \times \Sigma) \rightarrow Q$ *is the (possibly partial) transition function.*

Recognition paths of finite and infinite sequences are presented in the following:

1. A finite sequence σ such that $|\sigma| = n$ is recognizable by the Büchi automaton, if there exists a finite path of the form $q_0 \xrightarrow{\sigma[1]} q_1 \ldots q_{n-1} \xrightarrow{\sigma[n]} q_n$ where $\forall 0 \leq i \leq n.q_i \in Q.\forall 0 \leq i < n.\delta(q_i, \sigma[i+1]) = q_{i+1}$ and $q_n \in F$. Therefore, σ is recognizable by a finite path starting from the initial state q_0 and ending by a final state.
2. An infinite sequence σ is recognizable by the Büchi automaton, if there exists an infinite p path of the form $q_0 \xrightarrow{\sigma[1]} q_1 \ldots q_{n-1} \xrightarrow{\sigma[n]} q_n \xrightarrow{\sigma[n+1]} \ldots$ such that some final state f occurs infinitely often in p.

Proposition 5.20 *For any security automaton $A = \langle \Sigma, Q, q_0, \delta \rangle$ there exists a Büchi automaton recognizing the property A_P enforced by A.*

Proof: The Büchi automaton recognizing the property A_P enforced by *A* is the automaton $A' = \langle \Sigma, Q, q_0, Q, \delta \rangle$. This means that a security automaton is simply a Büchi automaton for which all states are finite states.

Definition 3.2, allows us to view edit automata characterizing effective$_=$-enforcers as sequence recognizers rather than sequence transformers. Although, edit automata have been introduced as sequence transformers, the main relevant contributions targeting EA-enforcement have been demonstrated using edit automata acting as effective$_=$-enforcers. Indeed, in [14] [13] [1], an effective$_=$-enforcer is characterized by an edit automaton that suppresses a sequence of potentially dangerous actions until it can confirm that the sequence is legal, at which point it inserts all the suppressed actions. This is exactly the same principle used by automata-based compilers. Following this intuition, we can easily construct a Büchi automaton specifying the property being enforced by an edit automaton acting as effective$_=$-enforcer.

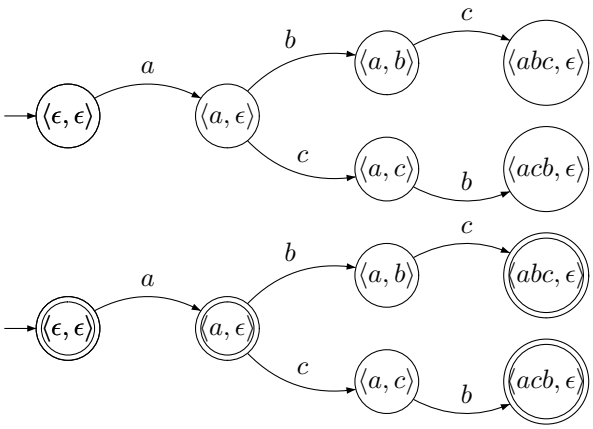

Figure 1. An edit automaton and the corresponding Büchi automaton.

Proposition 5.21 *For any edit automaton A such that $A = \langle \Sigma, \Sigma^* \times \Sigma^*, \langle \epsilon, \epsilon \rangle, \delta \rangle$ effectively$_=$-enforcing a property P, there exists a Büchi automaton specifying P.*

Proof: The Büchi automaton specifying the property *P* is $A' = \langle \Sigma, \Sigma^* \times \Sigma^*, \langle \epsilon, \epsilon \rangle, F, \delta' \rangle$ where:

- $F = \{\langle \sigma, \epsilon \rangle | \sigma \in P \cap \Sigma^*\}$ is the set of finite states.
- $\delta' : (Q \times \Sigma) \rightarrow Q$ is the transition function. For a state $q = \langle \sigma_{Acc}, \sigma_{Sup} \rangle$ and an input action *a*: $\delta'(q, a) =$ $\begin{cases} q' \text{ if } \delta(q, a) = (q', \tau). \\ \text{Undefined, if } \delta \text{ is not defined for the pair } (q, a). \end{cases}$

Figure 1 shows an edit automaton enforcing the property $P = \{a, abc, acb\}$ and the corresponding Büchi automaton.

6 Examples Of BHA-Enforceable Security Policies

In this section, we investigate *BHA*-enforceable security policies. We start by the security policies that can be derived from proposition 4.5. Then, we present two examples of *BHA*-enforceable practical security policies. The first is a class of *BSA*-enforceable policies and the second is a class of *BEA*-enforceable policies.

6.1 SHA-Enforceable Policies

Proposition 4.5 allows us to identify any *SHA*-enforceable property (when the actions set Σ is finite) as a *BHA*-enforceable property . We present briefly the four properties provided by Fong [9] as *SHA*-enforceable.

6.1.1 Chinese Wall Policy

The Chinese Wall policy [7] is an access control policy that defines the necessary rules to prevent conflict of interest. Conflict of interest can be characterized by accessing both the information of a party and the information of its competitor. To enforce this policy an execution monitor must check for each access whether the targeted information belongs to a party that is in conflict of interest with some party which some crucial information has been already disclosed to the accessing subject. To characterize this policy, the set of all subjects is defined by *S*, the set of all protected objects is defined by *O*, the set of all conflict of interest classes is defined by *T*, and each object *o* belongs to some conflict of interest class *t*. Since the order of access events is not needed to enforce the policy, Chinese wall policy is *SHA*-enforceable and by 4.5, its enforceable by some *k-BSA* if the subject set *S* and the object set *O* are finite and $|S \times O| = k$.

6.1.2 Low-Water-Mark Policy (for Subjects)

Low-Water-Mark Policy is defined by Biba in [5]. This policy defines the rules to be enforced within a system of entities where each entity can be either a subject or an object and to each entity *e* is assigned an integrity level *l(e)*. The set of objects is denoted by *O* and the set of subjects is denoted by *S*. The possible actions of the system are *read(s,o)*, *write(s,o)*, and *exec(s,o)* where s, s' are any two subjects, o, o' are any two objects. The set of all possible actions is defined by $\Sigma = \{read(s,o) | s \in S \wedge o \in O\} \cup \{exec(s,s') | s,s' \in S\} \cup \{write(s,o) | s \in S \wedge o \in O\}$. The three actions *read()*, *write()* and *exec()* obey to the following rules:

- *read(s,o)* is allowed without any constraint and affects the integrity level of s such that $l(s) \leftarrow l(s) \wedge l(o)$ where $\wedge$ is the greatest lower bound over integrity levels.
- *write(s,o)* is allowed if and only if $l(s) \geq l(o)$.
- *exec(s,s')* is allowed if and only if $l(s) \geq l(s')$.

Objects integrity levels are assigned once and thus are unchangeable while subjects integrity levels are affected by read actions. Since allowing any action depends only on the set of the already executed actions, this policy is *SHA*-enforceable and by consequence it is enforceable by some *k-BSA* if the set Σ is finite and $k = |\Sigma|$.

6.1.3 One-Out-Of-k Authorization

The One-Out-Of-k Authorization policy [11] specifies the access authorization rules by classifying programs into equivalence classes. Each equivalence class specifies a set of access authorizations that are granted to each program of that class. Whether one program belongs to a particular equivalence class depends on the actions performed by the program during execution. Once, a program is classified into some equivalence class, it can perform any action that is authorized for the class. An example of equivalence classes is provided in [11] where programs are classified into three classes: Browser, Editor and Shell. For example, if a program has opened a network socket, it is classified as a browser, and will be prevented from reading user files. This policy is *SHA*-enforceable and by consequence is enforceable by some *k-BSA* if the set of all possible actions Σ is a finite and $k = |\Sigma|$.

6.1.4 Assured Pipelines

Assured pipelines [25] [6] is a policy that ensures the integrity of data that are processed by pipelines of transformation procedures. This policy is defined for a set of data object *O* and a set of transformation procedures *S* where *create* is a special member of *S*. The set of possible actions is $S \times O$ which characterizes the application of transformation procedures to data objects. An assured pipelines policy is defined by an enabling relation $e \subseteq S \times S$ satisfying the two following constraints[9]:

- No circularity: the binary relation defines a directed acyclic graph (DAG).
- No pair of the form $\langle s, create \rangle$ may be included: create is the sole source node of the acyclic graph.

Intuitively, if a pair $\langle s, s' \rangle$ is in the relation *e* then any action $\langle s', o \rangle$ is allowed if and only if the last action performed on the object *o* is $\langle s, o \rangle$. According to [9], assured pipelines policy is enforceable by a *SHA* where the set of states is $2^{S \times O}$. By consequence, this policy is enforceable by some *k-BSA* if the set $S \times O$ is finite and $k = |S \times O|$.

6.2 Bounded Availability Policies

Bounded Availability Policies specify that any acquired resource must be released by some fixed point later in the execution. According to [20], a bounded availability policy is EM-enforceable if it is specified such that any resource can not be acquired more than some MWT (maximum waiting time) execution steps without being released. Enforcing such policies protects systems from denial of service attacks [10]. Figure 2 presents an example of a *BSA* used to enforce *Two-BA* security property, which is a bounded availability property. *Two-BA* ensures that each acquired resource must be released in at most 2 steps. The set of resources is $\{A, B\}$. Actions a and b represent acquiring resource A and B respectively, and actions $\overline{a}$ and $\overline{b}$ represent releasing resource A and B respectively. Action τ represents any action

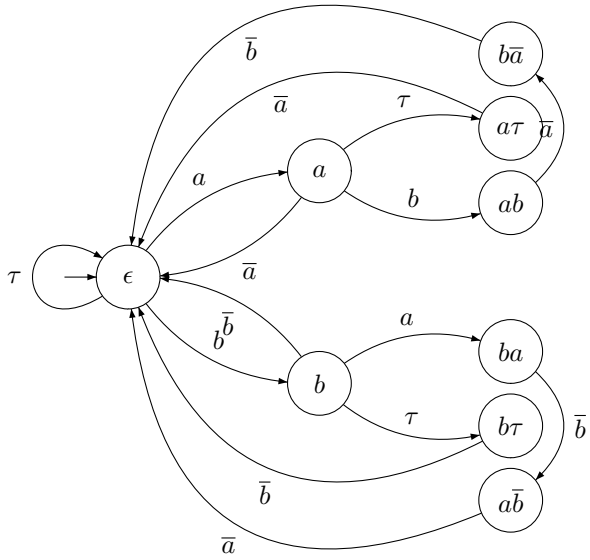

Figure 2. A bounded security automaton enforcing the *Two-BA* property.

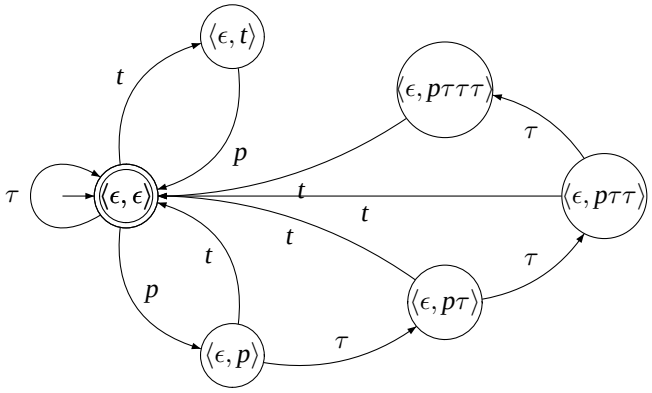

Figure 3. A bounded edit automaton enforcing a transaction-based property.

that is neither an action acquiring a resource nor an action releasing a resource. To enforce the policy, each execution must satisfy the following rules:

(1) At each execution point, no resource is acquired more than two computational steps.

(2) If for one execution point, a resource is taken during one computation step then the only action permitted by the automaton is the action releasing that resource. This is the case of states ba, $b\tau$, $a\tau$, ab, $b\bar{a}$, and $a\bar{b}$. For the other states, the automaton can take any τ action, any action acquiring a resource that is not already acquired, or any action releasing a resource that is already acquired. This is the case of states τ, a, and b.

The size of history needed to enforce this policy is two which is the bound defined by the bounded availability policy. The abstraction function β used to define the transition function is the following:

$$\begin{array}{lll} \alpha(a) = a & \alpha(a\tau) = a\tau & \alpha(a\bar{a}) = \epsilon \\ \alpha(ab) = ab & \alpha(a\tau\bar{a}) = \epsilon & \alpha(b) = b \\ \alpha(b\tau) = b\tau & \alpha(b\bar{b}) = \epsilon & \alpha(ba) = ba \\ \alpha(b\tau\bar{b}) = \epsilon & \alpha(ab\bar{a}) = b\bar{a} & \alpha(b\bar{a}\bar{b}) = \epsilon \\ \multicolumn{3}{c}{\alpha(ba\bar{b}) = a\bar{b} \quad \alpha(a\bar{b}\bar{a}) = \epsilon} \end{array}$$

6.3 Transaction-based Policies

Transaction-based properties specify that transactions must be atomic. A transaction is atomic if either the entire transaction is executed or no part of it is executed. To this class of properties belong database transactions [16] and e-commerce transactions. Transaction properties are usually specified by T^∞ where $T \subseteq \Sigma^*$ is the set of valid transactions defined over a set of possible actions Σ. A transaction property is not enforceable by security automata since there exists some illegal (bad) executions that can be extended to legal (valid) executions. An edit automaton can enforce a transaction-based property by suppressing all actions of the execution until reaching a complete transaction, at that moment the automaton insert the suppressed prefix. Since we are dealing with bounded history automata, constraints have to be imposed on the size of elements of *T*. Indeed, let $P = T^\infty$ be a transaction-based property, *P* is enforceable by a bounded edit automaton of bound *k* if and only if $T \subseteq \Sigma_k$.

Figure 3 represents a *BEA* enforcing the transaction-based property *P* defined by $P = \{tp, pt, p\tau t, p\tau\tau t, p\tau\tau\tau t\}^\infty$ where *t* is the action of taking a media resource, *p* is the action of paying for a media resource, and τ is any action other than taking or paying for a media resource. A transaction is accepted if it is either (1) taking a media resource and then paying immediately for it or (2) paying for a media resource and making at most three other actions before actually taking the media resource. The abstraction function β and the function α used to define the transition function are defined by the following:

$$\begin{array}{ll} \alpha(\beta(p)) = \alpha(p) = \langle\epsilon, p\rangle & \alpha(\beta(p\tau)) = \alpha(p\tau) = \langle\epsilon, p\tau\rangle \\ \alpha(\beta(tp)) = \alpha(\epsilon) = \langle\epsilon, \epsilon\rangle & \alpha(\beta(p\tau\tau t)) = \alpha(\epsilon) = \langle\epsilon, \epsilon\rangle \\ \alpha(\beta(t)) = \alpha(t) = \langle\epsilon, t\rangle & \alpha(\beta(p\tau\tau)) = \alpha(p\tau\tau) = \langle\epsilon, p\tau\tau\rangle \\ \multicolumn{2}{c}{\alpha(\beta(p\tau\tau\tau)) = \alpha(p\tau\tau\tau) = \langle\epsilon, p\tau\tau\tau\rangle} \end{array}$$

7 Conclusion and Future Work

In this paper, we propose a characterization of the security policies that are enforceable by execution monitors constrained by memory limitations. We characterize such memory limitation constraints by the size of memory used by an execution monitor to save the history of execution. The work presented here, is in the same line as the research work advanced by Schneider [20], Ligatti et.al [1, 13] and Fong [9] which addresses security policy enforcement. Our approach gives rise to a realistic evaluation of the enforcement power of execution monitoring. This evaluation is based on bounding the memory size used by the monitor to save execution history, and identifying the security policies enforceable under such constraint.

Our contribution is mainly threefold. First, we instantiate an abstraction based on memory limitation to security automata [20] as well as to edit automata [1]; the two main automata models characterizing EM-enforceable security policies. The result is a new class of automata that we call *bounded history automata* including two subclasses; *bounded security automata* and *bounded edit automata* characterizing security policies over finite and infinite executions. Second, we identify a new taxonomy of EM-enforceable properties that is directed by the size of the space used by execution monitors to save execution history. Third, we investigate the enforcement of locally testable properties by bounded history automata. Namely, we identify locally testable properties that are *BHA*-enforceable and show how to check whether a *BHA*-enforceable policy is locally testable.

As future work, we plan to investigate the panoply of results available on locally testable properties. More precisely, we will select the best algorithms identifying locally testable properties and adapt them to EM-enforceable properties. This will allow us to improve our *BHA* enforceable security policies classification by identifying new classes of real EM-enforceable policies.

References

[1] L. Bauer, J. Ligatti, and D. Walker. More enforceable security policies. In *Foundations of Computer Security*, pages 95–104, Copenhagen, Denmark, 25–26 July 2002. DIKU Technical Report.

[2] L. Bauer, J. Ligatti, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. Tech. Rep TR-681-03, Princeton University, May, 2003.

[3] L. Bauer, J. Ligatti, and D. Walker. Composing security policies with polymer. *SIGPLAN Not.*, 40(6):305–314, 2005.

[4] D. Beauquier and J. E. Pin. Languages and scanners. *Theoretical Comput. Sci.*, 84:3–21, 1991.

[5] K. Biba. Integrity considerations for secure computer systems. Technical Report 76372, US Air Force Electronic Systems Division, 1977.

[6] W. E. Boebert and R. Y. Kain. A practical alternative to hierarchical integrity policies. In *In Proceedings of the 8th National Computer Security Conference*, page 1827, October 1985.

[7] D. F. C. Brewer and M. J. Nash. The chinese wall security policy. In *IEEE Symposium on Security and Privacy*, pages 206–214, May 1989.

[8] P. Caron. Families of locally testable languages. *Theor. Comput. Sci.*, 242(1-2):361–376, 2000.

[9] P. L. Fong. Access control by tracking shallow execution history. In *In Proceedings of the 2004 IEEE Symposium on Security and Privacy*. Berkeley, California, May 2004.

[10] V. D. Gligor. A note on denial-of-service in operating systems. *IEEE Trans. Softw. Eng.*, 10(3):320–324, 1984.

[11] A. A. G. Edjladi and V. Chaudhary. History-based access control for mobile code. In *5th ACM Conference on Computer and Communications Security*, pages 38–48, San Francisco, CA, USA, November 1998.

[12] K. Hamlen, G. Morrisett, and F. Schneider. Computability classes for enforcement mechanisms. Technical Report TR2003-1908, Cornell University, 2003. To appear in ACM Transactions on Programming Languages and Systems.

[13] J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1–2):2–16, Feb. 2005. (Published online 26 Oct 2004.).

[14] J. Ligatti, L. Bauer, and D. Walker. Enforcing non-safety security policies with program monitors. Technical Report TR-720-05, Princeton University, Jan. 2005.

[15] A. Magnaghi and H. Tanaka. An efficient algorithm for order evaluation of strict locally testable languages.

[16] W. H. Paxton. A client-based transaction system to maintain data integrity. In *Proceedings of the seventh ACM symposium on Operating systems principles*, pages 18–23. ACM Press, 1979.

[17] D. Perrin and J. E. Pin. *Infinite Words. Automata, Semigroups, Logic and Games*, volume 141 of *Pure and Applied Mathematics*. Elsevier, 1004.

[18] D. Perrin and J. Pin. Semigroups and automata on infinite words, 1995.

[19] J.-E. Pin. Logic, semigroups and automata on words., 1994.

[20] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and Systems Security*, 3(1):30–50, 2000.

[21] R. M. S. Kim, R. McNaughton. A polynomial time algorithm for the local testability problem of deterministic finite automata. *IEEE Trans. Comput*, 40:1087–1093, 1991.

[22] R. M. S. Kim. Computing the order of a locally testable automaton. *SIAM Journal of Computing*, 23:1193–1215, 1994.

[23] A. Taivalsaari. JSR 139 J2ME Connected Limited Device Configuration 1.1, March 2003.

[24] A. N. Trahtman. An algorithm to verify local threshold testability of deterministic finite automata. *Lecture Notes in Computer Science*, 2214:164+, 2001.

[25] P. A. T. W. D. Young and W. E. Boebert. A verified labler for the secure ada target. In *In Proceedings of the 9th National Computer Security Conference*, page 5561, September 1986.