Execution Monitoring Enforcement Under Memory-Limitation Constraints

Chamseddine Talhi[†‡], Nadia Tawbi[‡], and Mourad Debbabi[†]

[‡] LSFM Group, Computer Science Department, Laval University, Quebec, (QC), Canada.
[†] Computer Security Laboratory, Concordia Institute for Information Systems Engineering, Concordia University, Montreal, (Qc), Canada.

Abstract. Recently, attention has been given to formally characterize security policies that are enforceable by different kinds of security mechanisms. 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, *Bounded History Automata*. Characterizing memory limitations leads us to define a precise taxonomy of security policies that are enforceable under memory limitation constraints.

***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. The efforts of some pioneer authors [21][12] [15] [8] 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) is a ubiquitous technique for security policies enforcement, this class of enforcement mechanisms has attracted the attention of the majority of researchers in this field. Execution monitors are enforcement mechanisms operating 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 [2].

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 research problem would guide the elaboration and the evaluation of lightweight security mechanisms for embedded platforms.

Fong [8] 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 properties over finite or infinite executions. The targeted properties are properties enforceable by security automata [21] and those enforceable by edit automata [1].

The remainder of this paper is organized as follows. We start by the related work in Section 2. In Section 3, we present the main definitions needed alongside the paper. Section 4 is dedicated to the presentation of the main characterizations of execution monitoring enforcement. Section 5 is devoted to the presentation of bounded history automata. In Section 6, we investigate EM-enforcement of locally testable properties. We present some examples of *BHA*-enforceable policies in Section 7 and we end by the conclusion and the future work in Section 8.

2 Related Work

Schneider [21] is the pioneer in characterizing EM-enforceable security policies. 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] [16] [15] 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. Having the power of modifying program actions at run time, edit automata are provably more powerful than security automata [15]. Hamlen et al. [12] provided an arithmetic hierarchy-based taxonomy of enforceable security policies. They investigated a larger set of enforcement mechanisms, including static enforcement, execution monitoring and program rewriting. This taxonomy leads to a more accurate characterization of EM-enforceable security policies.

Fong [8] provided a fine-grained, information-based characterization of EM-enforceable policies. In order to represent constraints on information available to execution monitors, he used abstraction functions over sequences of monitored programs. He defined a lattice on the space of all congruence relations over action sequences aimed at comparing classes of EM-enforceable security policies. The latter are limited to safety properties over finite executions. The investigated abstractions are (1) the mapping of action sequences into action sets and (2) the information stored in the Java execution Stack.

3 Definitions

We start by some notations of language theory. An alphabet Σ is a finite or infinite set of symbols representing program actions. In the sequel, we use interchangeably symbols and actions. The set of all finite sequences over Σ is denoted by Σ^*. The set of all infinite sequences is denoted by Σ^ω, and $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$ denotes the set of all finite and infinite sequences. 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' the concatenation of two languages L and L'. The difference of two languages L and L' is denoted by $L \setminus L'$. We denote by $\mid \sigma \mid$ 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, $\Sigma_{\leq k} = \{\sigma \in \Sigma^* : |\sigma| \leq k\}$ denotes the set of all possible sequences of length less than or equal to k. The set $(\Sigma_{\leq k} \times \Sigma_{\leq k})_{\leq k} = \{(\sigma, \sigma') \in \Sigma_{\leq k} \times \Sigma_{\leq k} : |\sigma\sigma'| \leq k\}$ 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 a 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 k length prefix of σ. Similarly, $\sigma[k+1..]$ denotes the suffix 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 $\sigma[i..i+k-1]$. The set of all k length factors of σ is denoted by $Fact_k(\sigma) = \{\sigma' \in \Sigma_k \mid \exists\, \sigma'' \in \Sigma^*.\exists \sigma''' \in \Sigma^\infty : \sigma = \sigma''\sigma'\sigma'''\}$. The set $Pref_{\leq k}$ is defined by $Pref_{\leq k}(\sigma) = \{\sigma' \in Pref(\sigma) :\mid \sigma' \mid \leq k\}$. Similarly, $Suf_{\leq k}(\sigma) = \{\sigma' \in Suf(\sigma) : \mid \sigma' \mid \leq k)\}$.

We need also some definitions related to security policies. 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: $\forall \sigma \in \Sigma^\infty.\, \sigma \in P \Rightarrow Pref(\sigma) \subseteq P$.

4 EM-Enforcement Characterization

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

4.1 Security automata

In this characterization, a monitor can intervene only by halting the program execution. According to this definition of EM-enforcement, Schneider [21] 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 1 (Security Automata). *A Security Automaton (SA)[21] 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.*

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 $q' = \delta(q, a)$. Let A_P denote the security policy enforced by a *BSA* A. Thus, A_P is the set of all, finite or infinite, sequences recognized 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.

4.2 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 2 (Edit Automata). *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 (possibly partial) transition function*[1]*.*

When given a current state q and an input action a, the transition function δ specifies a new state q' to enter and an actions sequence τ to execute. The executed sequence τ specifies the intervention action to take in order to enforce the property: (1) If $\tau =$ "a" then the input action a is to be accepted, (2) if

[1] The transition function definition presented here is equivalent to the original definition of Ligatti et al. [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 (1) to be closer to automata theory and (2) to facilitate automata construction in the proofs presentation.

$\tau \neq \epsilon \wedge \tau \neq$ "a" then the sequence τ is to be inserted, and (3) if $\tau = \epsilon$ then the input action a is to be suppressed. If the transition function is not defined for some state q and some action a, then the only possible intervention action to enforce the property is 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. Let $A(\sigma)$ denote the sequence edited by an edit automaton A while reading an input sequence σ and let A_P denote the property enforced by A. Since, *EA* can modify input sequences, they must obey to the two main principles:

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

EA ensure *Soundness* by transforming bad executions into valid executions, and ensure *transparency* by transforming valid executions into equivalent valid executions. Any edit automaton satisfying soundness and transparency is said to be an *effective* enforcer [1]. Let us denote by *effective*$_{\cong}$ *enforcement* the effective enforcement of edit automata based on a given equivalence relation $\cong$.

Definition 3 (Effective$_{\cong}$ Enforcement [1]). *Let A be an edit automaton and $\cong$ an equivalence relation over Σ^{∞}. The EA A 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][15] that edit automata acting as effective$_{=}$ enforcers are more powerful than security automata. 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] [16], [15]. *Renewal properties* is identified as a lower bound of properties that are effectively$_{=}$ enforceable by edit automata.

Definition 4. *A property P over Σ^{∞} is a renewal property if and only if it satisfies one of the two following conditions:*

$$\forall \sigma \in \Sigma^{\omega}.\ \sigma \in P \Leftrightarrow Pref(\sigma) \cap P \quad \text{is an infinite set} \tag{1}$$

$$\forall \sigma \in \Sigma^{\omega}.\sigma \in P \Leftrightarrow \forall \sigma' \in Pref(\sigma).\ \exists \sigma'' \in Pref(\sigma).\sigma' \in Pref(\sigma'') \wedge \sigma'' \in P \tag{2}$$

Proposition 1. *[15]* [2]

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

[2] This proposition corresponds to theorem 8 in [15].

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

Proof. Proposition 1 corresponds to theorem 8 in [15]. The reason behind presenting a detailed proof of this proposition is twofold: (1) adapting the proof provided in [15] to definition 2 and (2) explaining the ideas behind automata construction since they will be reused in many other other proofs that are presented in this paper.

The edit automaton A, effectively$_=$ enforcing P, is defined by $\langle \Sigma, Q, \langle \epsilon, \epsilon \rangle, \delta \rangle$ where:

- Σ is the set of finite or countably infinite input actions.
- $Q = \Sigma^* \times \Sigma^*$ is the set of finite or countably infinite automaton states. Each state is a pair $\langle \sigma_{Acc}, \sigma_{Sup} \rangle$ where $\sigma_{Acc}\sigma_{Sup}$ represents a finite sequence σ for which σ_{Acc} is the longest valid σ prefix i.e. the sequence edited by the automaton while reading σ, and σ_{Sup} is the suffix of σ that is suppressed by the automaton after reading σ. Note that $\sigma_{Sup} = \epsilon$ for any valid sequence σ.
- $q_0 \in Q$ is the initial state. It is the pair $\langle \epsilon, \epsilon \rangle$, which means that no prefix is accepted and no suffix is suppressed.
- $\delta : (Q \times \Sigma) \rightarrow (Q \times \Sigma^\infty)$ is the transition function. For a state $q = \langle \sigma_{Acc}, \sigma_{Sup} \rangle$ and an input action a, the state $q' = \delta(\langle \sigma_{Acc}, \sigma_{Sup} \rangle, a)$ is defined by:
$$q' = \begin{cases} (\langle \sigma_{Acc}, \sigma_{Sup} a \rangle, \epsilon) \text{ if } ((\sigma_{Acc}\sigma_{Sup} a \notin P) \wedge (\exists \sigma' \in \Sigma^* : \sigma_{Acc}\sigma_{Sup} a \sigma' \in P)) \\ (\langle \sigma_{Acc}\sigma_{Sup} a, \epsilon \rangle, \sigma_{Sup} a) \text{ if } (\sigma_{Acc}\sigma_{Sup} a \in P) \\ \text{Undefined, otherwise.} \end{cases}$$

The transition function ensures that only valid prefixes (satisfying P) will be edited by the automaton:

- For a finite input sequence $\sigma \in \Sigma^*$, if $\sigma \in P$ then $A(\sigma) = \sigma$ i.e. the entire sequence σ will be edited by A. If $\sigma \notin P$ then the automaton A edits the longest valid prefix of σ.
- For an infinite sequence $\sigma \in \Sigma^\infty$:
  - If $\sigma \in P$ then the automaton edits all the valid prefixes of σ. Since a valid sequence can count many invalid prefixes, for any invalid prefix $\sigma_1\sigma_2$ such that σ_1 is the longest valid prefix of $\sigma_1\sigma_2$, the automaton edits σ_1 and suppresses the sequence σ_2 in order to reinsert it when reaching the immediately next valid prefix $\sigma_1\sigma_2\sigma_3$.
  - If $\sigma \notin P$ then, by (2), there exists a longest invalid prefix σ' of σ for which any extension is an invalid sequence. The automaton ensures the edition of all the valid sequences i.e. all valid elements of $Pref(\sigma')$.

4.3 Fong's Characterization

Fong [8] has proposed an information-based approach characterizing EM-enforceable security policies according to the information consumed by execution monitors. 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 mapping different

sequences onto a single abstract state, the set of sequences that are visible to an execution monitor is reduced, and consequently, the set of security policies enforceable by that execution monitor is reduced. An abstraction function is defined according to the following definition:

Definition 5 (Abstraction Function[8]).
Let S be a finite or countably infinite set of abstract states and let α be any function such that $\alpha : \Sigma^* \rightarrow S$. The function α is an abstraction function if it satisfies the following compatibility property:

$$\forall w, w' \in \Sigma^*. \forall a \in \Sigma. \alpha(w) = \alpha(w') \Rightarrow \alpha(wa) = \alpha(w'a). \tag{3}$$

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

Definition 6 (α-SA[8]). *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[8]. Shallow history automata (SHA) is a special class of α-SA characterizing monitors tracking shallow access history. The information provided by a shallow access history determines the set of actions that have been previously granted. The formal definition of shallow history automata is the following.

Definition 7 (Shallow History Automata[8]). *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:*

$$\forall H \in 2^{\Sigma}. \forall a \in \Sigma.\ \delta(H, a) = \begin{cases} H \cup \{a\} \text{ if } H \text{ is a valid shallow history} & (a) \\ \text{Undefined, otherwise.} & (b) \end{cases}$$

5 Bounded History Automata

Bounded History Automata (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 encoding an action sequence of bounded length.
- For a bounded history h and an input action a, the image h' (if it is defined), given by the transition function, is an abstraction of the sequence ha.

5.1 Bounded Security Automata

Definition 8 (Bounded Security Automata). *A BSA of bound k (k-BSA) is a SA* $\langle \Sigma, Q = \Sigma_{\leq k}, q_0, \delta \rangle$ *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 reads an input action a, if there exists a state h' such that $h' = \delta(h, a)$ then h' is an abstraction of the history ha. This means that only the abstraction h' of ha is relevant for the enforcement of the security policy A_P in any extension of ha. Thus, the transition function δ defines an abstraction function $\beta : \Sigma_{\leq k+1} \rightarrow \Sigma_{\leq k}$ where $\delta(h, a) = \beta(ha)$. We denote the abstraction function defined by the transition function of a *BHA* A by A_β. 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.

Theorem 1. *For any two positive integers k and k' such that* $k < k'$*, we have* $EM_{kSA} \subset EM_{k'SA}$*.*

Proof. First, we prove that any property of EM_{kSA} can be enforced by a *k'-BSA*. Second, we prove that there exists a property in $EM_{k'SA}$ that cannot be enforced by any *k-BSA*:

(1) Let P be a property of EM_{kSA}. Then, there exists a *k-BSA* A enforcing P such that $A = \langle \Sigma, Q = \Sigma_{\leq k}, q_0, \delta \rangle$. The *k'-BSA* enforcing P is $A' = \langle \Sigma, Q' = \Sigma_{\leq k'}, q_0, \delta' \rangle$ where $\delta' : (\Sigma_{\leq k'} \times \Sigma) \rightarrow \Sigma_{\leq k'}$ is defined such that $\forall q \in \Sigma_{\leq k'}. \forall a \in \Sigma. \delta'(q, a) = \delta(q, a)$ if δ is defined for q and a and is not defined otherwise.
(2) There exists a property $P \in EM_{k'SA}$ for which, there is no *k-BSA* enforcing it. This property is defined by $P = \{Pref(a_1 \dots a_{k'+1})\}$ where the set of input actions is defined by $\Sigma = \{a_1, \dots, a_{k'+1}\}$. Indeed, to recognize the sequence $a_1 \dots a_{k'+1}$ we need to save the history of the last k' actions which is not possible by any *k-BSA* since $k < k'$.

The three following propositions explain the connection between *BSA* and Fong's α*-SA* and *SHA*.

Proposition 2. *For any BSA* $A = \langle \Sigma, Q = \Sigma_{\leq k}, q_0, \delta \rangle$*, there exist an* α*-SA enforcing* A_{P_f}*.*

Proof. The α*-SA* enforcing A_{P_f} is defined by the *SA* $\langle \Sigma, Q' = Q \cup \{bad\}, \alpha(\epsilon), \delta' \rangle$ where:

- The abstract states set Q' contains all the states of Q with the addition of a new state *bad* representing any bad sequence (not belonging to A_{P_f}).
- The transition function $\delta' : (Q' \times \Sigma) \rightarrow Q'$ is defined such that for any state q and any input action a, $\delta'(q,a) = \delta(q,a)$ if δ is defined for the pair (q,a). Otherwise, δ' is not defined.
- The abstraction function $\alpha : \Sigma^* \rightarrow Q$ is defined by the following:
  $\forall \sigma \in \Sigma^*.\alpha(\sigma) = \begin{cases} \delta^*(\sigma) \text{ if } \sigma \in A_{P_f} \\ \text{bad , otherwise.} \end{cases}$

  The abstraction function α satisfies the compatibility property (3). Indeed, for any action $a \in \Sigma$ and any two finite sequences $\sigma, \sigma' \in \Sigma^*$ such that $\alpha(\sigma) = \alpha(\sigma') = h$, we have the two following cases:

  - If $h = bad$ then $\sigma, \sigma' \notin A_{P_f}$ and by consequence $\sigma a, \sigma' a \notin A_{P_f}$ and $\alpha(\sigma a) = \alpha(\sigma' a) = bad$.
  - If $h \neq bad$ then σ and σ' are in A_{P_f}. If there exists some state h' for which $\delta(h,a) = h'$ then σa and $\sigma' a$ are in A_{P_f} and are reachable by two paths ending in the state h' and by consequence $\alpha(\sigma a) = \alpha(\sigma' a) = h'$. If there is no state h' for which $\delta(h,a) = h'$ then σa and $\sigma' a$ are not in A_{P_f} and by consequence $\alpha(\sigma a) = \alpha(\sigma' a) = bad$.

Proposition 3. *If the set of abstract states Q is finite, then 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.*

Proof. If the set of states Q is finite, then the following statement is true:

$$\exists k' \in \mathbf{N}^+ . \exists \beta' : Q \rightarrow \Sigma_{\leq k'} . \forall q, q' \in Q . q \neq q' \Rightarrow \beta'(q) \neq \beta'(q') \tag{4}$$

Let k be the smallest positive integer satisfying 4, and let β be any mapping function satisfying 4. The existence of k and β is guaranteed since the set of abstract states Q is finite. Indeed, k can take any value that is greater than or equal to $|Q|$.

The *k-BSA* enforcing P is defined by the *SA* $A' = \langle \Sigma, \Sigma_{\leq k}, \beta(\alpha(\epsilon)), \delta' \rangle$ where the transition function $\delta' : (Q' \times \Sigma) \rightarrow Q'$ is defined such that for any abstract state q and any input action a, $\delta'(\beta(q), a) = \delta(q,a)$ if δ is defined for the pair (q,a). Otherwise, δ' is not defined.

It is obvious that the *α-SA* A and the *k-BSA* A' enforce the same property since A and A' define the same automaton under state renaming.

Corollary 1. *If the set of input actions Σ is finite such that $|\Sigma| = k$, then, for any SHA enforcing a property P, there exists a k-BSA enforcing P.*

Proof. This result can be directly derived from proposition 3. Let $A = \langle \Sigma, 2^{\Sigma}, \emptyset, \delta \rangle$ be the shallow history automaton enforcing the property P. If Σ is finite then 2^{Σ}, which is the set of all states of A, is finite and by proposition 3, there exists a *BSA* enforcing P.

The *k-BSA* enforcing P is defined by $\langle \Sigma, \Sigma_{\leq k}, \epsilon, \delta' \rangle$ where the transition function δ' is defined by the following:

$$\forall \sigma \in \Sigma_{\leq k}. \forall a \in \Sigma. \delta'(\sigma, a) = \begin{cases} \sigma \text{ if } a \in Act(\sigma) & (a) \\ \sigma a \text{ if } a \notin Act(\sigma) \wedge \delta(Act(\sigma), a) = Act(\sigma) \cup \{a\} & (b) \\ \text{Undefined, otherwise.} & (c) \end{cases}$$

where $Act : \Sigma_{\leq k} \rightarrow 2^{\Sigma}$ is the function that returns for each sequence σ the set of actions present in σ. Notice that the length of the sequence resulting from rule (b) is always less than or equal to k, since we allow at most one occurrence of an action 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*.

5.2 Bounded Edit Automata

A bounded history, used for the definition of a bounded edit automaton (*BEA*), 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 *BEA*, we use the construction technique adopted in [15].

Definition 9. *A BEA of bound k (k-BEA) is an EA* $\langle \Sigma, Q = (\Sigma_{\leq k} \times \Sigma_{\leq k})_{\leq k}, q_0, \delta \rangle$ *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* $\langle \sigma_{Acc}, \sigma_{Sup} \rangle$ *such that* $\sigma_{Acc}\sigma_{Sup} \in \Sigma_{\leq k}$.
- 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 $\beta : \Sigma_{\leq k+1} \rightarrow \Sigma_{\leq k}$ 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 $\alpha : \Sigma_{\leq k} \rightarrow (\Sigma_{\leq k} \times \Sigma_{\leq k})_{\leq k}$ specifies the accepted sequence and the suppressed one depending on the property being enforced by the *BEA*. Let EM_{kEA} denote the set of *k-BEA*-enforceable properties.

Theorem 2. *For any two positive integers k and k' such that* $k < k'$*, we have* $EM_{kEA} \subset EM_{k'EA}$.

Proof. This proposition can be easily proved by following the same intuition used to prove theorem 1.

First, we prove that any property of EM_{kEA} can be enforced by a *k'-BEA*. Second, we prove that there exists a property in $EM_{k'EA}$ that cannot be enforced by any *k-BEA*:

(1) Let P be a property of EM_{kEA}. Then, there exists a *k-BEA* A enforcing P such that $A = \langle \Sigma, Q = (\Sigma_{\leq k} \times \Sigma_{\leq k})_{\leq k}, q_0, \delta \rangle$. The *k'-BEA* enforcing P is $A' = \langle \Sigma, Q = (\Sigma_{\leq k'} \times \Sigma_{\leq k'})_{\leq k'}, q_0, \delta' \rangle$ where the transition function $\delta' : ((\Sigma_{\leq k'} \times \Sigma_{\leq k'})_{\leq k'} \times \Sigma) \rightarrow (\Sigma_{\leq k'} \times \Sigma_{\leq k'})_{\leq k'}$ is defined such that: $\forall q \in (\Sigma_{\leq k'} \times \Sigma_{\leq k'})_{\leq k'}. \forall a \in \Sigma. \delta'(q, a) = \delta(q, a)$ if δ is defined for q and a, otherwise $\delta'(q, a)$ is not defined.

(2) There exists a property $P \in EM_{k'EA}$ for which, there is no *k-BEA* enforcing it. This property is defined by $P = \{a_1 \dots a_{k'+1}\}$ where the set of input actions is defined by $\Sigma = \{a_1, \dots, a_{k'+1}\}$. Indeed, to recognize the sequence $a_1 \dots a_{k'+1}$ we need to save the history of the last *k'* actions which is not possible by any *k-BEA* since $k < k'$.

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

Theorem 1 and Theorem 2 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 Theorem 1, we get the following taxonomy: $EM_{0SA} \subset EM_{1SA} \subset EM_{2SA} \dots \subset EM_{SA}$. 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 having 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 Theorem 2, we get the following taxonomy: $EM_{0EA} \subset EM_{1EA} \subset EM_{2EA} \dots \subset EM_{EA}$. Note that for any positive integer k, we have $EM_{kSA} \subset EM_{kEA}$.

6 Bounded History Automata and Local Testability

In the following, we investigate the connection between *local* properties and *BHA*-enforceable properties. Local properties are properties that are recognizable by *scanners*; automata equipped with a finite memory and a *sliding* window of a fixed length n [3]. 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. Some results of this section are devoted to identify prefix-closed locally testable properties. This will help us to identify locally testable properties that are *BSA*-enforceable.

6.1 Locally Testable Properties

The main class of local properties is the class of *Locally Testable* (*LT*) properties. In the sequel, we present the formal definition of *LT* properties and the different classes that can derived from it.

Definition 10 (Locally Testable Properties). [4]

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

$$\forall \sigma \in \Sigma^*.\sigma \in L \Leftrightarrow (\sigma \in X) \vee ((\sigma[..k-1] \in P) \wedge (\sigma[|\sigma|-k+2..] \in S) \wedge (Fact_k(\sigma) \subseteq F)) \quad (5)$$

$$\forall \sigma \in \Sigma^\omega.\sigma \in L \Leftrightarrow \forall \sigma' \in Pref(\sigma).\exists \sigma'' \in Pref(\sigma).\sigma' \in Pref(\sigma'') \wedge \sigma'' \in L \quad (6)$$

where $\sigma[|\sigma|-k+2..]$ is the suffix of σ of length $k-1$.

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 all sequences of a locally testable property L is defined by $L = ((P\Sigma^\infty \cap (\Sigma^* S)^\infty) \setminus \Sigma^* \overline{F} \Sigma^\infty) \cup X$ where $\overline{F} = \Sigma^k \setminus F$. The set of all finite sequences of L is defined by $L \cap \Sigma^* = ((P\Sigma^* \cap \Sigma^* S) \setminus \Sigma^* \overline{F} \Sigma^*) \cup X$. Similarly, the set of all 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 $X = \{\sigma \in L : |\sigma| < k\}$. Thus, a sequence $\sigma \in \Sigma_{\leq k}$ is a sequence of L if and only if $\sigma \in X$. The sets P and S define two sets of factors, $F_{initial}$ and $F_{terminal}$ where:

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

The set $F_{initial}$ 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.

The property $LL = ((\{ab\}\Sigma^* \cap \Sigma^*\{ba\}) \setminus \Sigma^*\{aaa, abb, bab, bba, bbb\}\Sigma^*) \cup \{aa, bb\}$ is an example of a *3-LT* property where $\Sigma = \{a, b, c\}$, $P = \{ab\}$, $S = \{ba\}$, $X = \{aa, bb\}$, $F = \{aba, baa, aab\}$, $\overline{F} = \{aaa, abb, bab, bba, bbb\}$, 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.

[4] 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 [7] [19].

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:

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

Proposition 4 (Prefix-closed Locally Testable Properties).
Let k be any positive integer, and let $F \subseteq \Sigma_k$, $P, S \subseteq \Sigma_{k-1}$, and $X \subseteq \Sigma_{\leq k-1}$ be the sets used to define a k-LT property L. The property L is prefix-closed if and only if:

(I) $X \cup F_{initial}$ *is prefix-closed, and*
(II) $F_R \subseteq F_{terminal}$.

Proof. Since we have an equivalence statement, we have to prove the two implication directions:

- *If direction:* We prove that if (I) and (II) are satisfied, then L is prefix-closed. The property L can be written as $L = L' \cup L''$ where $L' = X \cup F_{initial} = L \cap \Sigma_{\leq k}$ and $L'' = \{\sigma \in L : \sigma \text{ is infinite or } |\sigma| > k\}$ and $L' \cap L'' = \emptyset$.
  If (I) is satisfied i.e. if $X \cup F_{initial}$ is prefix-closed, then L' is prefix-closed. If (II) is satisfied, then any factor of F_R can used as a suffix of a sequence of L. For any sequence σ of L'', any prefix σ' of σ such that $|\sigma'| > k$ is a sequence of L because σ' is of the form $\sigma'' f$ where $\sigma'' \in \Sigma^+$ and $f \in F_{terminal}$. In addition, the set of all prefixes of σ of length less than or equal to k is a subset of L because this set is equal to $Pref(f')$ where $f' \in F_{initial}$ and by (I) this set is prefix-closed. We conclude that if (I) and (II) are satisfied then L is prefix-closed.

- *Only-If direction:* We prove that if L is prefix-closed then conditions (I) and (II) are satisfied. We proceed by contradiction:
  1. Let suppose that (I) is not satisfied i.e. $X \cup F_{initial}$ is not prefix-closed. Then we can find a sequence $f\sigma \in L$ such that $\exists \sigma' \in \Sigma^*.\sigma' \in Pref(f) \wedge \sigma' \notin L$ where $\sigma \in \Sigma^*$ and $f \in F_{initial}$. Consequently L is not prefix-closed (Contradiction).
  2. Let suppose that (II) is not satisfied i.e. $\neg(F_R \subseteq F_{terminal})$. Then we can find some factor f such that $f \in F_R$ and $f \notin F_{terminal}$. According to the definition of F_R, we have either (a) $\exists \sigma \in \Sigma^*.\sigma f \in L$ or (b) $\exists \sigma \in \Sigma'.\exists \sigma' \in \Sigma^+.\sigma f \sigma' \in L$. If (a) is true then f must be in $F_{terminal}$ because the sequences of L can not end with a factor that is not in $F_{terminal}$. If (b) is true then the sequence $\sigma f \sigma'$ is in L while its prefix σf is not in L. Consequently L is not prefix-closed (Contradiction).

Among LT properties, we can identify four main classes: *prefix testable* (PT), *suffix testable* (ST), *prefix-suffix testable* (PST), and *strongly locally testable* (SLT) properties [19]. We start by *prefix testable* properties that are recognizable by inspecting only prefixes of limited size.

Definition 11 (Prefix Testable Properties). *Let k be a positive integer. A property L of Σ^∞ is k-prefix testable (k-PT) if there exist two sets $P \subseteq \Sigma_k$ and $X \subseteq \Sigma_{\leq k-1}$ such that the elements of L are defined by the following rule:*

$$\forall \sigma \in \Sigma^\infty . \sigma \in L \Leftrightarrow (\sigma \in X) \vee (\sigma[..k] \in P). \tag{7}$$

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

According to this definition, The set of all sequences of a prefix testable property L is defined by $L = P\Sigma^\infty \cup X$. The set of all finite sequences of L is defined by $L \cap \Sigma^* = P\Sigma^* \cup X$. Similarly, the set of all infinite sequences of L is defined by the set $L \cap \Sigma^\omega = P\Sigma^\omega$.

Proposition 5 (Prefix-closed Prefix Testable Properties).
Let k be any positive integer, and let L be a k-prefix-testable property defined by the two sets $P \subseteq \Sigma_k$ and $X \subseteq \Sigma_{\leq k-1}$. The property L is prefix-closed if and only if $X \cup P$ is prefix-closed.

Proof. Since we have an equivalence statement, we have to prove the two implication directions:

- *If direction:* We suppose that $X \cup P$ is prefix-closed and we prove that any sequence of L has all its prefixes in L. The property L can be written as $L = X \cup P\Sigma^\infty$. For any $\sigma \in L$ we have two cases:
  1. $|\sigma| \leq k$. Obviously, $\sigma \in X \cup P$. Since $X \cup P$ is prefix-closed then we have $\mathit{Pref}(\sigma) \subseteq (X \cup P) \subset L$.
  2. The sequence σ is infinite or $|\sigma| > k$. The sequence σ is of the form $p\sigma'$ where $p \in P$ and $\sigma' \in \Sigma^\infty$. Any prefix of σ of length greater than k is in L because it is of the form $p\sigma''$ where $\sigma'' \in \Sigma^+$. Any prefix of σ of length less than or equal to k is in L because it is an element of $\mathit{Pref}(P)$.

- *Only-If direction:* We prove that if L is prefix-closed then $X \cup P$ is prefix-closed. We proceed by contradiction. Let suppose that $X \cup P$ is not prefix-closed. Then $\exists \sigma \in X \cup P . \exists \sigma' \in \mathit{Pref}(\sigma) . \sigma' \notin X \cup P$. We have $\sigma \in L$ and $\sigma' \notin L$ because $\sigma' \notin X$ and $\sigma' \notin P\Sigma^\infty$. Consequently L is not prefix-closed (Contradiction).

Respectively, *suffix testable* properties are recognizable by inspecting only suffixes of limited size.

Definition 12 (Suffix Testable Properties). *Let k be a positive integer. A property L of Σ^∞ is k-suffix testable (k-ST) if there exist two sets $S \subseteq \Sigma_k$ and $X \subseteq \Sigma_{\leq k-1}$ such that the elements of L are defined by the two following rules:*

$$\forall \sigma \in \Sigma^* . \sigma \in L \Leftrightarrow (\sigma \in X) \vee (\sigma[|\sigma| - k + 1..] \in S) \tag{8}$$

$$\forall \sigma \in \Sigma^{\omega}.\sigma \in L \Leftrightarrow \forall \sigma' \in Pref(\sigma).\, \exists \sigma'' \in Pref(\sigma).\, \sigma' \in Pref(\sigma'') \wedge \sigma'' \in L \quad (9)$$

where $\sigma[|\sigma| - k + 1..]$ *is the suffix of* σ *of length* k*.*

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 all sequences of a suffix testable property L is defined by $L = (\Sigma^* S)^{\infty} \cup X$. The set of all finite sequences of L is defined by $L \cap \Sigma^* = \Sigma^* S \cup X$. Similarly, the set of all 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 13 (Prefix-Suffix Testable Properties). *Let* k *be a positive integer. A property* L *of* Σ^{∞} *is k-prefix-suffix testable (k-PST) if there exist three sets* $P, S \subseteq \Sigma_k$ *and* $X \subseteq \Sigma_{\leq k-1}$ *such that the elements of* L *are defined by the two following rules:*

$$\forall \sigma \in \Sigma^*.\, \sigma \in L \Leftrightarrow \sigma \in X \vee (\sigma[..k] \in P \wedge \sigma[|\sigma - k + 1..] \in S) \quad (10)$$

$$\forall \sigma \in \Sigma^{\omega}.\, \sigma \in L \Leftrightarrow \forall \sigma' \in Pref(\sigma).\, \exists \sigma'' \in Pref(\sigma).\sigma' \in Pref(\sigma'') \wedge \sigma'' \in L \quad (11)$$

where $\sigma[|\sigma| - k + 1..]$ *is the suffix of* σ *of length* k*.*

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 all sequences of a prefix-suffix testable property L is defined by $L = (P\Sigma^{\infty} \cap (\Sigma^* S)^{\infty}) \cup X$. The set of all finite sequences of L is defined by the set $(L \cap \Sigma^*) = P\Sigma^* S \cup X$. Similarly, the set of all infinite sequences of L is defined by the set $L \cap \Sigma^{\omega} = P\Sigma^{\omega} \cap (\Sigma^* S)^{\omega}$.

Proposition 6. *Let* k *be any positive integer, and let* P, S, X *be three sets such that* $P, S \subseteq \Sigma_k$*, and* $X \subseteq \Sigma_{\leq k-1}$*. 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.*

Proof. Since we can extend a sequence $\sigma \notin L$ to a sequence $\sigma' \in L$, the property L is not prefix-closed. Indeed, such sequence σ can be extended to a sequence σs where $s \in S$ is any suffix of the suffixes used to define L.

The *strongly locally* testable properties are a variety of LT properties that are recognizable by inspecting only factors of fixed size.

Definition 14 (Strongly Locally Testable Properties). *Let k be a positive integer. A property L of Σ^∞ is k-strongly locally testable (k-SLT) if there exist a set $F \subseteq \Sigma_k$ such that the elements of L are defined by the following rule:*

$$\forall \sigma \in \Sigma^\infty . \sigma \in L \Leftrightarrow \forall \sigma' \in Pref(\sigma). Fact_k(\sigma') \subseteq F \tag{12}$$

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 all sequences of a strongly locally testable property L is defined by $L = \Sigma^\infty \setminus (\Sigma^* F \Sigma^\infty)$. The set of all finite sequences of L is defined by the set $(L \cap \Sigma^*) = \Sigma^* \setminus (\Sigma^* F \Sigma^*)$. Similarly, the set of all infinite sequences of L is defined by the set $L \cap \Sigma^\omega = \Sigma^\omega \setminus (\Sigma^* F \Sigma^\omega)$.

Proposition 7 (Prefix-closed Strongly Locally Testable Properties). *Let k be any positive integer, and let $F \subseteq \Sigma_k$. If L is a k-strongly-locally testable property defined by the set F, then L is prefix-closed.*

Proof. The definition of a k-strongly-locally testable property L makes no constraints on the sequences of $\Sigma_{\leq k-1}$. Therefore the prefix-closed set $\Sigma_{\leq k-1}$ is a subset of L. In addition, any sequence σ of L such that $|\sigma| \geq k$ has all its factors in F. Indeed, any prefix σ' of σ such $|\sigma'| \geq k$ has all its factors in F and consequently $\sigma' \in L$. Any prefix σ' of σ such that $|\sigma'| < k$ is an element of $\Sigma_{\leq k-1}$ and consequently $\sigma' \in L$. We conclude that L is prefix-closed.

6.2 *BSA*-Enforceable Locally Testable Properties

In what follows, we investigate locally testable properties that are enforceable by *BSA*. Since *BSA* is a class of security automata, a locally testable property has to be prefix-closed in order to be *BSA*-enforceable. The following propositions identify *BSA*-enforceable *LT* properties and *LT* properties that are not *BSA*-enforceable.

Theorem 3. *Let k be any positive integer. Any prefix-closed k-prefix testable property L that is defined according to definition 11 is enforceable by some k-BSA.*

Proof. From proposition 5, L is prefix-closed if and only if $X \cup P$ is prefix-closed. The k-*BSA* enforcing L is defined by $A = \langle \Sigma, Q = \Sigma_{\leq k}, \epsilon, \delta \rangle$ where the transition function δ is defined by the following:

$$\forall \sigma \in Q. \forall a \in \Sigma.\ \delta(\sigma, a) = \begin{cases} \sigma a \text{ if } \sigma a \in P \cup X & (1) \\ \sigma \text{ if } \sigma \in P & (2) \\ \text{Undefined, otherwise.} & (3) \end{cases}$$

The k-*BSA* A recognizes all the sequences σ of L and only the sequences of L. We consider the two cases:

1. *Case* $|\sigma| \leq k$, *i.e.* $\sigma \in P \cup X$: Since $P \cup X$ is prefix-closed and each sequence of $P \cup X$ is represented by a state, rule (1) ensures that each sequence of $P \cup X$ is recognizable by A. Indeed, any sequence σ of $P \cup X$ is recognizable by the path: $\epsilon \xrightarrow{\sigma[1]} \sigma[1] \dots \sigma[m-1] \xrightarrow{\sigma[m]} \sigma$ where $|\sigma| = m$. It is easy to see that no sequence of $\Sigma_{\leq k} \setminus (X \cup P)$ can be recognized by A.
2. Case $|\sigma| > k$: Rules (1) and (2) ensure that σ is recognizable by A. Indeed any finite prefix $\sigma'\sigma''$ of σ where $\sigma' \in P$ and $\sigma'' \in \Sigma^\infty$ is recognizable by the path $\epsilon \xrightarrow{\sigma'[1]} \sigma'[1] \dots \sigma'[m-1] \xrightarrow{\sigma'[m]} \sigma' \xrightarrow{\sigma''[1]} \sigma' \dots \sigma' \xrightarrow{\sigma''[d]} \sigma'$ where $|\sigma'| = m$ and $|\sigma''| = d$. By ensuring that any sequence starts by a prefix of P, no sequence that is not in L can be recognized by A.

Theorem 4. *Let k be any positive integer. Any k-strongly locally testable property L that is defined according to definition 14 is enforceable by some k-BSA.*

Proof. The *BSA* enforcing L is defined by $A = \langle \Sigma, Q = \Sigma_{\leq k}, \epsilon, \delta \rangle$ where the transition function δ is defined by:

$$\forall \sigma \in Q. \forall a \in \Sigma.\ \delta(\sigma, a) = \begin{cases} \sigma a \text{ if } \sigma a \in \Sigma_{\leq k-1} \vee \sigma a \in F & (1) \\ \sigma[2..]a \text{ if } \sigma \in F \wedge \sigma[2..]a \in F & (2) \\ \text{Undefined, otherwise.} & (3) \end{cases}$$

The transition function δ allow the automaton to recognize any sequence of length less than k and any sequence of length greater than or equal to k that have all its factors in F. Let σ be any sequence of Σ^∞. We have the two following cases:

1. *Case* $|\sigma| < k$: Since, the property L makes no constraint on sequences of length less than k, then any sequence σ of $\Sigma_{\leq k-1}$ is recognizable by the path $\epsilon \xrightarrow{\sigma[1]} \sigma[1] \dots \sigma[m-1] \xrightarrow{\sigma[m]} \sigma$ where $|\sigma| = m$.
2. *Case* σ infinite or $|\sigma| \geq k$: Since L accepts only the sequences of length greater than or equal to k that have all their factors of length k in F, rules (1) and (2) ensure that any finite prefix $\sigma'\sigma''$ of σ where $|\sigma'| = k-1$, is recognizable by A by the path $\epsilon \xrightarrow{\sigma'[1]} \sigma'[1] \dots \xrightarrow{\sigma'[k-1]} \sigma' \xrightarrow{\sigma''[1]} f_1 \xrightarrow{\sigma''[2]} f_2 \dots f_{d-1} \xrightarrow{\sigma''[d]} f_d$ where $|\sigma''| = d$ and $f_1 = \sigma'[2..]\sigma''[1]$ and $\forall\, 1 < i \leq d. f_i = f_{i-1}[2..]\sigma''[i]$ and $\forall\, 1 \leq i \leq d. f_i \in F$. Thus all the factors of length k of σ are in F.

Theorem 5. *Let k be any positive integer. Any prefix-closed k-locally testable property L that is defined according to Definition 10 is enforceable by some k-BSA.*

Proof. From Proposition 4, L is prefix-closed if and only if: (I) $X \cup F_{initial}$ is prefix-closed, and (II) $F_R \subseteq F_{terminal}$. The *$k$-BSA* A enforcing L is defined by the 5-tuple $\langle \Sigma, Q = \Sigma_{\leq k}, k, \epsilon, \delta \rangle$ where the transition function δ is defined by:

$$\forall \sigma \in Q. \forall a \in \Sigma.\ \delta(\sigma, a) = \begin{cases} \sigma a \text{ if } \sigma a \in X \cup F_{initial} & (a) \\ \sigma[2..]a \text{ if } (\sigma \in F_R \wedge \sigma[2..]a \in F_R) & (b) \\ \text{Undefined, otherwise.} & (c) \end{cases}$$

The *k-BSA* A recognizes all the sequences of L and only the sequences of L. Let σ be any sequence of Σ^∞. We have the two following cases:

1. Case $|\sigma| \leq k$, *i.e.* $\sigma \in X \cup F_{initial}$: Since $X \cup F_{initial}$ is prefix-closed and by the definition of Q, each sequence of $X \cup F_{initial}$ is represented by a state, rule (a) ensures that each sequence of $X \cup F_{initial}$ is recognizable by A. Indeed, any sequence σ of $X \cup F_{initial}$ is recognizable by the path: $\epsilon \xrightarrow{\sigma[1]} \sigma[1] \ldots \xrightarrow{\sigma[m]} \sigma$ where $|\sigma| = m$. It is clear that any sequence of $\Sigma_{\leq k} \setminus (X \cup F_{initial})$ can not be recognized by A.
2. Case σ infinite or $|\sigma| > k$: Any sequence of L of length greater than k is a sequence that starts by a factor of $F_{initial}$, all its factors are in F_R and ends by a factor of $F_{terminal}$. By definition, σ starts by a factor of $F_{initial}$. The definition of Q and rule (b) ensure that each prefix of σ with length greater than or equal to k is recognizable and ends by a factor belonging to F_R. By condition (II), any factor of F_R is in $F_{terminal}$, which means that all finite prefixes that are recognized by A are in L since they end by a factor of $F_{terminal}$. Indeed any finite prefix $\sigma'\sigma''$ of σ is recognizable by the path $\epsilon \xrightarrow{\sigma'[1]} \sigma'[1] \ldots \sigma'[k-1] \xrightarrow{\sigma'[k]} \sigma' \xrightarrow{\sigma''[1]} f_1 \xrightarrow{\sigma''[2]} f_2 \ldots f_{d-1} \xrightarrow{\sigma''[d]} f_d$ where $\sigma' \in F_{initial}$, $|\sigma''| = d$, $f_1 = \sigma'[2..]\sigma''[1]$, $\forall\ 1 < i \leq d. f_i = f_{i-1}[2..]\sigma''[i]$, $\forall\ 1 \leq i \leq d. f_i \in F_R$, and $f_d \in F_{terminal}$. It is obvious that any sequence that is not in L can not be recognized by A.

Theorem 6. *Suffix testable properties and prefix-suffix testable properties are not BSA-enforceable.*

Proof. The proof can be immediately deduced from Proposition 6. Indeed, suffix testable properties and prefix-suffix testable properties are not prefix-closed and consequently they are not enforceable by security automata.

6.3 *BEA*-Enforceable Locally Testable Properties

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

Theorem 7. *Let k be any positive integer. Any k-prefix testable property, that is defined according to Definition 11, is enforceable by some k-BEA.*

Proof. Let k be any positive integer and let $P \subseteq \Sigma_{\leq k}$ and $X \subseteq \Sigma_{\leq k-1}$ be the sets used to define the k-prefix testable property L over Σ^∞. The *k-BEA* enforcing L is the *EA* $\langle \Sigma, Q = (\Sigma_{\leq k} \times \Sigma_{\leq k})_{\leq k}, \langle \epsilon, \epsilon \rangle, \delta \rangle$ where the transition function δ is defined by the following:

$$\forall \langle \sigma_{Acc}, \sigma_{Sup} \rangle \in Q. \forall a \in \Sigma.$$
$$\delta(\langle \sigma_{Acc}, \sigma_{Sup} \rangle, a) = \begin{cases} (\langle \sigma_{Acc}\sigma_{Sup}a, \epsilon \rangle, \sigma_{Sup}a) \text{ if } \sigma_{Acc}\sigma_{Sup}a \in P \cup X & (1) \\ (\langle \sigma_{Acc}, \sigma_{Sup}a \rangle, \epsilon) \text{ if } \sigma_{Acc}\sigma_{Sup}a \in \Sigma_{\leq k} \setminus (P \cup X) & (2) \\ (\langle \sigma_{Acc}, \epsilon \rangle, a) \text{ if } (\sigma_{Acc} \in P \wedge \sigma_{Sup} = \epsilon) & (3) \\ \text{Undefined, otherwise.} \end{cases}$$

The definition of A ensures that any sequence of L is recognizable by A and that no sequence that is not in L can be recognized by A. For any sequence σ of Σ^∞, we have the two following cases:

1. Case $|\sigma| \leq k$, i.e. $\sigma \in P \cup X$: The definition of δ ensures that any sequence $\sigma \in P \cup X$ can be recognized by reaching a state $\langle \sigma, \epsilon \rangle$. Rules (1) and (2) ensure that any such state is reachable by the path $\langle \epsilon, \epsilon \rangle \xrightarrow[\tau_1]{\sigma[1]} \langle \sigma_{1\,Acc}, \sigma_{1\,Sup} \rangle \xrightarrow[\tau_2]{\sigma[2]} \dots \xrightarrow[\tau_{m-1}]{\sigma[m-1]}$ $\langle \sigma_{m-1\,Acc}, \sigma_{m-1\,Sup} \rangle \xrightarrow[\tau_m]{\sigma[m]} \langle \sigma, \epsilon \rangle$ where:
   $|\sigma| = m$ and $\forall i.1 < i < m.\tau_i = \delta(\langle \sigma_{i-1\,Acc}, \sigma_{i-1\,Sup} \rangle, \sigma[i])$, and $\forall i.1 \leq i < m$, the state $\langle \sigma_{i\,Acc}, \sigma_{i\,Sup} \rangle$ satisfies: $\sigma_{i\,Acc}\sigma_{i\,Sup} = \sigma[..i] \wedge (\sigma[..i] \in P \cup X \Rightarrow \sigma_{i\,Sup} = \epsilon)$.
   Rule (1) ensures that the entire read sequence is edited when it is in $X \cup P$. When the read sequence is in $Pref(X \cup P)$ but not in $X \cup P$, rule (1) ensures that the longest valid prefix is edited and rule (2) ensures that the remaining suffix is suppressed. When reaching a valid sequence, the longest suppressed suffix is edited by rule (1) generating a valid sequence.
2. Case $|\sigma| > k$, i.e. $\sigma = \sigma'\sigma''$ where $\sigma' \in P$ and $\sigma'' \in \Sigma^\infty$: Rules (1) and (2) allow the recognition of the prefix σ' by reaching the state $\langle \sigma', \epsilon \rangle$, and rule (3) allows reading the actions of σ'' by looping in the state $\langle \sigma', \epsilon \rangle$. If $\sigma'' \in \Sigma^*$, then σ is recognizable by the path $\langle \epsilon, \epsilon \rangle \xrightarrow[\tau_1]{\sigma'[1]} \dots \xrightarrow[\tau_m]{\sigma'[m]} \langle \sigma', \epsilon \rangle \xrightarrow[\sigma''[1]]{\sigma''[1]} \langle \sigma', \epsilon \rangle \dots \langle \sigma', \epsilon \rangle \xrightarrow[\sigma''[d]]{\sigma''[d]} \langle \sigma', \epsilon \rangle$ where $|\sigma'| = m$, $|\sigma''| = d$, and the path recognizing the prefix σ' is presented in the previous case. If $\sigma'' \in \Sigma^\omega$, then after editing σ', the path recognizing σ loops infinitely while reading and editing the elements of σ''.

Intuitively, the elements of $P \cup X$ are recognizable by reading the prefixes of $P \cup X$ and editing only those that are elements of $P \cup X$. After reaching a valid prefix $\sigma' \in P$, the automaton can recognize any extension $\sigma = \sigma'\sigma''$ of σ' by looping in the state $\langle \sigma', \epsilon \rangle$ while reading and editing the sequence σ without suppressing any action of σ''. It is obvious that any sequence that is not in L can not be recognized by the automaton A.

Theorem 8. *Let k be any positive integer. Any k-strongly locally testable property L that is defined according to Definition 14 is enforceable by some k-BEA.*

Proof. Let k be any positive integer and let $F \subseteq \Sigma_k$ be the set used to define the k-strongly locally testable property L over Σ^∞. The *k-BEA* enforcing L is defined by $A = \langle \Sigma, Q = (\Sigma_{\leq k} \times \Sigma_{\leq k})_{\leq k}, \langle \epsilon, \epsilon \rangle, \delta \rangle$ where the transition function $\delta : \{\langle \sigma, \epsilon \rangle | \sigma \in \Sigma_{\leq k}\} \times \Sigma \to \{\langle \sigma, \epsilon \rangle | \sigma \in \Sigma_{\leq k}\} \times \Sigma^*$ is defined by the following:

$$\forall \langle \sigma, \epsilon \rangle \in Q. \forall a \in \Sigma.\ \ \delta(\langle \sigma, \epsilon \rangle, a) = \begin{cases} (\langle \sigma a, \epsilon \rangle, a) \text{ if } \sigma a \in \Sigma_{\leq k-1} \cup F & (1) \\ (\langle \sigma[2..]a, \epsilon \rangle, a) \text{ if } \sigma, \sigma[2..]a \in F & (2) \\ Undefined, otherwise. \end{cases}$$

This *BEA* recognizes any sequence σ of L. We have two cases:

1. Case $|\sigma| < k$: The definition of δ ensures that any sequence σ of length less than k can be recognized by reaching the state $\langle \sigma, \epsilon \rangle$ after editing the whole sequence σ. If $|\sigma = m|$, then rules (1) ensures that any such state is reachable by the path $\langle \epsilon, \epsilon \rangle \xrightarrow[\sigma[1]]{\sigma[1]} \langle \sigma[..1], \epsilon \rangle \xrightarrow[\sigma[2]]{\sigma[2]} \dots \xrightarrow[\sigma[m-1]]{\sigma[m-1]} \langle \sigma[..m-1], \epsilon \rangle \xrightarrow[\sigma[m]]{\sigma[m]} \langle \sigma, \epsilon \rangle$.
2. Case $|\sigma| \geq k$: By rule (1) and rule (2), any sequence having all its factors of length k in F is recognizable. By definition 14, σ is in the property L if and only if all prefixes of σ are in L. Any prefix of σ of length less than k is recognizable by a path as explained in case 1. Any prefix of length greater than or equal to k is of the form $\sigma' = f\sigma''$ where $f \in F$ and $\sigma'' \in \Sigma^*$. The prefix σ' is recognizable by the path: $\langle \epsilon, \epsilon \rangle \xrightarrow[f[1]]{f[1]} \langle f[..1], \epsilon \rangle \xrightarrow[f[2]]{f[2]} \dots \xrightarrow[f[k-1]]{f[k-1]}$ $\langle f[..k-1], \epsilon \rangle \xrightarrow[f[k]]{f[k]} \langle f, \epsilon \rangle \xrightarrow[\sigma''[1]]{\sigma''[1]} \langle f_1, \epsilon \rangle \xrightarrow[\sigma''[2]]{\sigma''[2]} \dots \xrightarrow[\sigma''[m-1]]{\sigma''[m-1]} \langle f_{m-1}, \epsilon \rangle \xrightarrow[\sigma''[m]]{\sigma''[m]}$ $\langle f_m, \epsilon \rangle$ where for all integer i such that $1 \leq i \leq m. f_i \in F \wedge f_i \in Suf(f\sigma''[..i])$.

From what follows, it is obvious that any sequence that is not in L can not be recognized by A.

Theorem 9. *Let k be any positive integer. Any k-locally testable property L that is defined according to Definition 10 is enforceable by some k-BEA.*

Proof. We prove this result by constructing the *BEA* enforcing L. Let k be any positive integer and let $P, S \subseteq \Sigma_{k-1}$, $X \subseteq \Sigma_{\leq k-1}$ and $F \subseteq \Sigma_k$ be the sets used to define the k-locally testable property L over Σ^∞. The *BEA* enforcing L is defined by the *EA* $A = \langle \Sigma, Q = (\Sigma_{\leq k} \times \Sigma_{\leq k})_{\leq k}, \langle \epsilon, \epsilon \rangle, \delta \rangle$ of bound k where the partial transition function δ is defined by the following:

$\forall \langle \sigma_{Acc}, \sigma_{Sup} \rangle \in Q. \forall a \in \Sigma.\ \sigma_{Acc}\sigma_{Sup} \in Pref(X \cup P) \Rightarrow$

$$\delta(\langle \sigma_{Acc}, \sigma_{Sup} \rangle, a) = \begin{cases} (\langle \sigma_{Acc}, \sigma_{Sup} a \rangle, \epsilon) \text{ if } \sigma_{Acc}\sigma_{Sup}a \in (\Sigma_{\leq k-1} \setminus X) \cup (F_{initial} \setminus F_{terminal}) & (1) \\ (\langle \sigma_{Acc}\sigma_{Sup}a, \epsilon \rangle, \sigma_{Sup}a) \text{ if } \sigma_{Acc}\sigma_{Sup}a \in X \cup (F_{initial} \cap F_{terminal}) & (2) \\ \text{Undefined, otherwise.} \end{cases}$$

$\forall \langle \sigma_{Acc}, \sigma_{Sup} \rangle \in Q. \forall a \in \Sigma.\ \sigma_{Acc}\sigma_{Sup} \in F \Rightarrow$

$$\delta(\langle \sigma_{Acc}, \sigma_{Sup} \rangle, a) = \begin{cases} (\langle \sigma_{Acc}[2..]\sigma_{Sup}a, \epsilon \rangle, \epsilon) \text{ if } \sigma_{Acc}[2..]\sigma_{Sup}a \in F_{terminal} & (3) \\ (\langle \sigma_{Acc}[2..], \sigma_{Sup}a \rangle, \sigma_{Sup}a) \text{ if } \sigma_{Acc}[2..]\sigma_{Sup}a \in F \setminus F_{terminal} & (4) \\ \text{Undefined, otherwise.} \end{cases}$$

The automaton A recognizes any finite sequence σ of L, we consider the two cases:

1. Case $|\sigma| \leq k$, i.e. $\sigma \in X \cup (F_{initial} \cap F_{terminal})$: The definition of δ ensures that any element of the set $X \cup (F_{initial} \cap F_{terminal})$ can be recognized by reaching a state $\langle \sigma, \epsilon \rangle$. Rules (1) and (2) ensure that any such state is reachable by the path $\langle \epsilon, \epsilon \rangle \xrightarrow[\tau_1]{\sigma[1]} \langle \sigma_{1\,Acc}, \sigma_{1\,Sup} \rangle \xrightarrow[\tau_2]{\sigma[2]} \dots \xrightarrow[\tau_{m-1}]{\sigma[m-1]} \langle \sigma_{m-1\,Acc}, \sigma_{m-1\,Sup} \rangle \xrightarrow[\tau_m]{\sigma[m]}$ $\langle \sigma, \epsilon \rangle$ where $|\sigma| = m$ and $\forall i.1 < i < m. \tau_i = \delta(\langle \sigma_{i-1\,Acc}, \sigma_{i-1\,Sup} \rangle, \sigma[i])$, and $\forall i. 1 \leq i < m.\ \ \sigma_{i\,Acc}\sigma_{i\,Sup} = \sigma[..i] \wedge (\sigma[..i] \in X \cup (F_{initial} \cap F_{terminal}) \Rightarrow \sigma_{i\,Sup} = \epsilon)$.

2. Case $|\sigma| > k$, i.e. $\sigma = \sigma'\sigma''$ where $\sigma' \in F_{initial}$ and $Fact_k(\sigma'\sigma'') \subseteq F$ and σ ends by a factor of $F_{terminal}$: Rules (1) and (2) allow the recognition of the prefix σ' by reaching the state $\langle \sigma'_{Acc}, \sigma'_{Sup} \rangle$ where $\sigma'_{Sup} = \epsilon$ if $\sigma' \in F_{terminal}$. Rules (3) and (4) ensure that any factor of length k of σ is an element of F and that the last factor is an element of $F_{terminal}$. Indeed σ is recognizable by the path $\langle \epsilon, \epsilon \rangle \xrightarrow[\tau'_1]{\sigma'[1]} \langle {\sigma'}_{1\,Acc}, {\sigma'}_{1\,Sup} \rangle \xrightarrow[\tau'_2]{\sigma'[2]} \dots \xrightarrow[\tau'_{m-1}]{\sigma[m-1]}$ $\langle {\sigma'}_{m-1\,Acc}, {\sigma'}_{m-1\,Sup} \rangle \xrightarrow[\tau'_m]{\sigma'[m]} \langle {\sigma'}_{Acc}, {\sigma'}_{Sup} \rangle \xrightarrow[\tau''_1]{\sigma''[1]} \longrightarrow \langle f_{1\,Acc}, f_{1\,Sup} \rangle \xrightarrow[\tau''_2]{\sigma''[2]}$ $\dots \xrightarrow[\tau''_{d-1}]{\sigma''[d-1]} \langle f_{d-1\,Acc}, f_{d-1\,Sup} \rangle \xrightarrow[\tau''_d]{\sigma''[d]} \longrightarrow \langle f_d, \epsilon \rangle$ where:
- $|\sigma'| = k$ and $|\sigma''| = d$.
- $\forall i.1 < i \leq k. \tau'_i = \delta(\langle {\sigma'}_{i-1\,Acc}, {\sigma'}_{i-1\,Sup} \rangle, \sigma'[i])$
- $\forall i.1 \leq i \leq k.\ {\sigma'}_{i\,Acc}{\sigma'}_{i\,Sup} = \sigma'[..i] \wedge (\sigma'[..i] \in X \cup (F_{initial} \cap F_{terminal}) \Rightarrow {\sigma'}_{i\,Sup} = \epsilon)$
- $f_{1\,Acc}f_{1\,Sup} = \sigma'[2..]\sigma''[1]$.
- $\forall\, 1 \leq i \leq d. f_{i\,Acc}f_{i\,Sup} \in F \wedge (f_{i\,Acc}f_{i\,Sup} \in F_{terminal} \Rightarrow (f_{i\,Acc} \in F_{terminal} \wedge f_{i\,Sup} = \epsilon)$.

By definition 10, a locally testable property is a renewal property. Therefore, any valid infinite sequence σ of L is recognizable by the bounded history automaton. Any such valid infinite sequence σ is a sequence that starts by a prefix of P and has all its factors of length k in F by alternating between factors of $F_{terminal}$ and factors of $F \setminus F_{terminal}$. Any valid prefix of σ is recognizable following the path construction described above. Any invalid prefix σ' of σ can be read by the automaton by outputting the longest valid prefix (ending by a factor of $F_{terminal}$) and reaching the state $\langle f_{Acc}, f_{Sup} \rangle$ where $f \in F \setminus F_{terminal}$ is the last factor of σ'. By definition 10, any invalid prefix σ' can be extended to some valid prefix σ'' which can be recognized by a finite path as described above.

Theorem 10. *Suffix testable properties are not BEA-enforceable.*

Proof. We have to prove that for any suffix testable property, there is no *BEA* enforcing it. We proceed by contradiction. For some positive integer k, let P be any k-suffix testable property defined by the two sets $S \subseteq \Sigma_k$ and $X \subseteq \Sigma_{\leq k-1}$. Let us suppose that there exists a *BEA* $A = \langle \Sigma, Q \subseteq (\Sigma_{\leq k'} \times \Sigma_{\leq k'})_{\leq k'}, q_0, \delta \rangle$ of bound *k'* enforcing *P*. However, we can find a sequence $\sigma \in P$ that is not recognizable by *A*. Such sequence can be any sequence $\sigma = \sigma' s$ such that s is any suffix from S and $\sigma' \in \Sigma^*$ is any sequence satisfying (1) $\mid \sigma' \mid > k'$ and (2) $s \notin Fact_{|s|}(\sigma')$. Intuitively, in order to recognize σ, we need to suppress the entire prefix σ' and save it in the bounded history and reinsert it after identifying the suffix *s*. This is not possible since the size of σ' is greater than the size of the bounded history that the automaton can track. Therefore, we have proved that there is no *BEA* enforcing *P*.

Theorem 11. *Prefix-suffix testable properties are not BEA-enforceable.*

Proof. For any k-prefix-suffix testable property L defined by the three sets $P \subseteq \Sigma_k$, $S \subseteq \Sigma_k$ and $X \subseteq \Sigma_{\leq k-1}$, we suppose that there exists a k'-*BEA* A enforcing L. We can find a sequence $\sigma \in L$ that is not recognizable by A. Such sequence can be any sequence $\sigma = p\sigma' s$ where s is any suffix from S, p is any prefix from P and $\sigma' \in \Sigma^*$ is any sequence satisfying (1) $\mid \sigma' \mid > k'$ and (2) $\forall s' \in S. s' \notin Fact_{\mid s' \mid}(\sigma')$. The recognition of σ requires the suppression of the entire prefix $p\sigma'$ and saving it in the bounded history in order to reinsert it after identifying the suffix s. This is not possible since the size of σ' is greater than the size of the bounded history that the automaton can track. Therefore, we have proved that there is no *BHA* enforcing L.

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

Theorem 12. *Let k be any positive integer. Any k-suffix testable property L that is defined according to Definition 12 is enforceable by some edit automaton.*

Proof. We prove this result by constructing the edit automaton enforcing L. The *EA* enforcing L is defined by $\langle \Sigma, \Sigma^* \times \Sigma^*, \langle \epsilon, \epsilon \rangle, \delta \rangle$ where the transition function δ is defined by the following:
$\forall \langle \sigma_{Acc}, \sigma_{Sup} \rangle \in \Sigma^* \times \Sigma^*. \forall a \in \Sigma.$

$$\delta(\langle \sigma_{Acc}, \sigma_{Sup} \rangle, a) = \begin{cases} (\langle \sigma_{Acc}, \sigma_{Sup} a \rangle, \epsilon) \text{ if } \sigma_{Acc}\sigma_{Sup} a \notin X \wedge Suf(\sigma_{Acc}\sigma_{Sup} a) \cap S = \emptyset & (1) \\ (\langle \sigma_{Acc}\sigma_{Sup} a, \epsilon \rangle, \sigma_{Sup} a) \text{ if } \sigma_{Acc}\sigma_{Sup} a \in X \vee Suf(\sigma_{Acc}\sigma_{Sup} a) \cap S \neq \emptyset & (2) \end{cases}$$

Theorem 13. *Let k be any positive integer. Any k-prefix-suffix testable property L that is defined according to Definition 13 is enforceable by some edit automaton.*

Proof. We prove this result by constructing the edit automaton enforcing L. The *EA* enforcing L is defined by $\langle \Sigma, \Sigma^* \times \Sigma^*, \langle \epsilon, \epsilon \rangle, \delta \rangle$ where the transition function δ is defined by the following:
$\forall \langle \sigma_{Acc}, \sigma_{Sup} \rangle \in \Sigma^* \times \Sigma^*. \forall a \in \Sigma.$

$$\delta(\langle \sigma_{Acc}, \sigma_{Sup} \rangle, a) = \begin{cases} (\langle \sigma_{Acc}, \sigma_{Sup} a \rangle, \epsilon) \text{ if } \sigma_{Acc}\sigma_{Sup} a \notin X \wedge \\ (Pref(\sigma_{Acc}\sigma_{Sup} a) \cap P = \emptyset \vee Suf(\sigma_{Acc}\sigma_{Sup} a) \cap S = \emptyset) & (1) \\ (\langle \sigma_{Acc}\sigma_{Sup} a, \epsilon \rangle, \sigma_{Sup} a) \text{ if } \sigma_{Acc}\sigma_{Sup} a \in X \vee \\ (Pref(\sigma_{Acc}\sigma_{Sup} a) \cap P \neq \emptyset \wedge Suf(\sigma_{Acc}\sigma_{Sup} a) \cap S \neq \emptyset) & (2) \end{cases}$$

6.4 Locally Testable EM-Enforceable Properties

In 6.2 and 6.3, we have demonstrated a strong connection between BHA-enforceable properties and locally testable 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 locally testable properties. Deciding whether a property is locally testable has been well investigated and many deciding algorithms where proposed [25] [23] [22] [17]. 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[5] are deterministic automata. First we recall the formal definition of Büchi automata and explain their property recognition mode.

Definition 15 (Büchi Automata[20]). *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.*

Recognizing 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 path p 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 8. *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 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 [15] [16] [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.

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

Proposition 9. *For any edit automaton $A = \langle \Sigma, \Sigma^* \times \Sigma^*, \langle \epsilon, \epsilon \rangle, \delta \rangle$ 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.

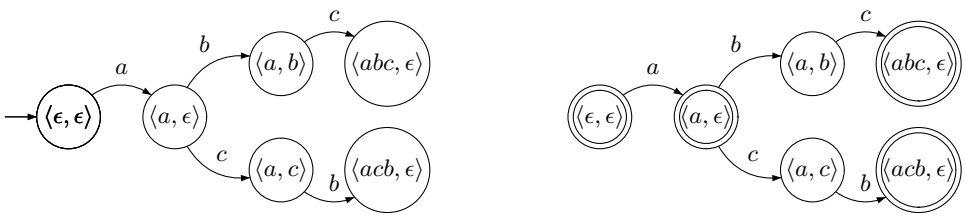

Fig. 1. An edit automaton and the corresponding Büchi automaton.

7 Examples Of BHA-Enforceable Security Policies

In this section, we present some *BHA*-enforceable security policies. We start by the security policies that can be derived from corollary 1. 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.

7.1 SHA-Enforceable Policies

Corollary 1 allows us to identify any *SHA*-enforceable property (when the actions set Σ is finite) as a *BHA*-enforceable property . We present briefly the properties provided by Fong in [8].

Chinese Wall Policy The Chinese Wall policy [6] 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 for 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 $o \in O$ belongs to some conflict of interest class $t \in T$. Since the order of access events is not needed to enforce the policy, Chinese wall policy is *SHA*-enforceable [8] and by Corollary 1, it is enforceable by some *k-BSA* if the subject set S and the object set O are finite and $|S \times O| = k$.

Low-Water-Mark Policy (for Subjects) Low-Water-Mark Policy is defined by Biba in [4]. 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 modifies the integrity level of as follows: $l(s) \leftarrow l(s) \wedge l(o)$ 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 can be modified by read actions. Since allowing any action depends only on the set of the already executed actions, this policy is *SHA*-enforceable [8] and by consequence it is enforceable by some *k-BSA* if the set Σ is finite and $k = |\Sigma|$.

One-Out-Of-k Authorization Policy 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 [8] and consequently is enforceable by some *k-BSA* if the set of all possible actions Σ is finite such that $k = |\Sigma|$.

Assured Pipelines Policy Assured pipelines [27] [5] 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 objects 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 [8]:

- 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 [8], assured pipelines policy is enforceable by a *SHA* where the set of states is $2^{S \times O}$. Consequently, this policy is enforceable by some *k-BSA* if the set $S \times O$ is finite and $k = |S \times O|$.

7.2 Bounded Availability Policies

Bounded availability policies specify that any acquired resource must be released by some fixed point later in the execution. According to [21], 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 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, $b\overline{a}$, and $a\overline{b}$. 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:

$$\alpha(a) = a \qquad \alpha(a\tau) = a\tau \qquad \alpha(a\overline{a}) = \epsilon \qquad \alpha(ab) = ab \qquad \alpha(a\tau\overline{a}) = \epsilon$$
$$\alpha(b) = b \qquad \alpha(b\tau) = b\tau \qquad \alpha(b\overline{b}) = \epsilon \qquad \alpha(ba) = ba \qquad \alpha(b\tau\overline{b}) = \epsilon$$
$$\alpha(ab\overline{a}) = b\overline{a} \qquad \alpha(b\overline{a}\overline{b}) = \epsilon \qquad \alpha(ba\overline{b}) = a\overline{b} \qquad \alpha(a\overline{b}\overline{a}) = \epsilon$$

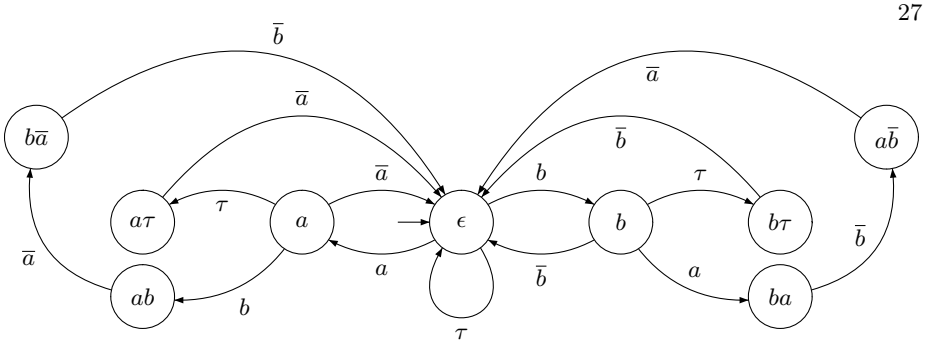

Fig. 2. A bounded security automaton enforcing the *Two-BA* property.

7.3 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 [18] 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 $T \subseteq \Sigma_{\leq k}$.

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:

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

8 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. The work

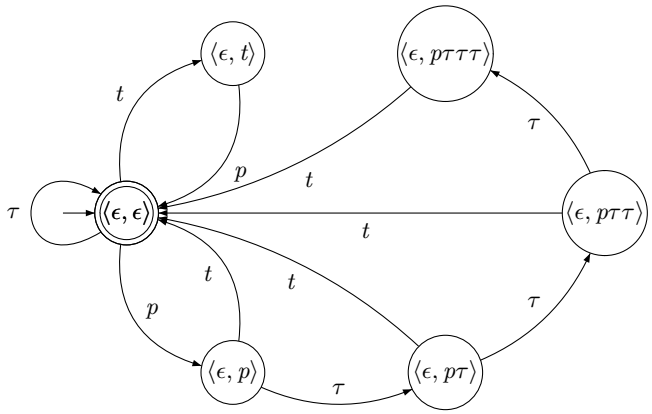

Fig. 3. A bounded edit automaton enforcing a transaction-based property.

presented here, is in the same line as the research work advanced by Schneider [21], Ligatti et.al [1, 16] and Fong [8] 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 [21] 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 an EM-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. Lujo Bauer, Jarred Ligatti, and David Walker. More enforceable security policies. In *Foundations of Computer Security*, pages 95–104, Copenhagen, Denmark, 25–26 July 2002.

2. L. Bauer, J. Ligatti, and D. Walker. Composing security policies with polymer. *SIGPLAN Not.*, 40(6):305–314, 2005.
3. D. Beauquier and J. E. Pin. Languages and scanners. *Theoretical Computer Science*, 84:3–21, 1991.
4. K. Biba. Integrity considerations for secure computer systems. Technical Report 76372, US Air Force Electronic Systems Division, 1977.
5. 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.
6. 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.
7. Pascal Caron. Families of locally testable languages. *Theor. Comput. Sci.*, 242(1-2):361–376, 2000.
8. P.W. 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.
9. Cédric Fournet and Andrew D. Gordon. Stack inspection: theory and variants. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 307–318. ACM Press, 2002.
10. Virgil D. Gligor. A note on denial-of-service in operating systems. *IEEE Trans. Softw. Eng.*, 10(3):320–324, 1984.
11. A. Acharya 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. Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. Computability classes for enforcement mechanisms. *ACM Trans. Program. Lang. Syst.*, 28(1):175–205, 2006.
13. Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions of Software Engineering*, 3(2):125–143, 1977.
14. Jay Ligatti, Lujo Bauer, and David Walker. Enforcing non-safety security policies with program monitors. In *Computer Security—ESORICS 2005: 10th European Symposium on Research in Computer Security*, volume 3679 of *Lecture Notes in Computer Science*, pages 355–373, September 2005.
15. J. Ligatti, Lujo. Bauer, and D. Walker. Enforcing non-safety security policies with program monitors. Technical Report TR-720-05, Princeton University, January 2005.
16. 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, February 2005. (Published online 26 Oct 2004.).
17. Antonio Magnaghi and Hidehiko Tanaka. An efficient algorithm for order evaluation of strict locally testable languages.
18. William 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.
19. D. Perrin and J. E. Pin. *Infinite Words. Automata, Semigroups, Logic and Games*, volume 141 of *Pure and Applied Mathematics*. Elsevier, 1004.
20. J. E. Pin. Finite semigroups and recognizable languages : an introduction. In J. Fountain, editor, *NATO Advanced Study Institute Semigroups, Formal Languages and Groups*, pages 1–32. Kluwer academic publishers, 1995.
21. F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and Systems Security*, 3(1):30–50, 2000.

22. R. McCloskey S. Kim, R. McNaughton. A polynomial time algorithm for the local testability problem of deterministic finite automata. *IEEE Trans. Comput*, 40:1087–1093, 1991.
23. R. McNaughton S. Kim. Computing the order of a locally testable automaton. *SIAM Journal of Computing*, 23:1193–1215, 1994.
24. A. Taivalsaari. Java Specification Request 139 J2ME Connected Limited Device Configuration 1.1, March 2003.
25. A. N. Trahtman. An algorithm to verify local threshold testability of deterministic finite automata. *Lecture Notes in Computer Science*, 2214:164+, 2001.
26. Mahesh Viswanathan. *Foundations for the Run-time Analysis of Software Systems*. PhD thesis, University of Pennsylvania, 2000.
27. P. A. Telega 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.