Available online at www.sciencedirect.com

ScienceDirect

Information and Computation 206 (2008) 158–184

Information and Computation

www.elsevier.com/locate/ic

Execution monitoring enforcement under memory-limitation constraints

Chamseddine Talhi [b,a,*], Nadia Tawbi [a], Mourad Debbabi [b]

[a] *LSFM Group, Computer Science Department, Laval University, Quebec, QC, Canada*
[b] *Computer Security Laboratory, Concordia Institute for Information Systems Engineering, Concordia University, Montréal, QC, Canada*

Received 10 November 2006; revised 2 May 2007
Available online 28 November 2007

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 [25,12,19,9] contribute to the emergence of a new research field that targets characterizing enforcement mechanisms and identifying the classes of enforceable security policies. Since execution monitoring (EM) 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].

[*] Corresponding author.
Email addresses: Chamseddine.Talhi@ift.ulaval.ca, talhi@ciise.concordia.ca (C. Talhi), Nadia.Tawbi@ift.ulaval.ca (N. Tawbi), debbabi@ciise.concordia.ca (M. Debbabi).
URLs: http://www.ift.ulaval.ca/, http://www.ciise.concordia.ca/

0890-5401/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.ic.2007.07.009

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 memory-constrained systems (e.g., embedded platforms). Fong [9] is the first one who presented an interesting attempt to answer this research problem. He presented a general theoretical framework to characterize security policies that are enforceable by execution monitors constrained by the available information about the execution history. However, the results of Fong are limited to prefix-closed security policies over finite executions and do not provide precise answers about the memory cost of security enforcement.

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 security policies over finite or infinite executions. The targeted policies are those that are specified by security automata (*SA*) [25] and those that are specified by edit automata (*EA*) [1]. We introduce *bounded history automata* (BHA) as subclasses of SA and EA to characterize execution monitors using bounded memory to track the execution history.

An important contribution of this work is the investigation of locally testable properties enforcement. Locally testable properties [3] are classes of languages where recognizing whether a sequence σ belongs to a property P is based on checking σ-subsequences of bounded size. We provide complete results defining the connection between locally testable properties and BHA-enforceable properties. Also, we define a general approach to identify security policies that can be enforced by monitors tracking bounded execution histories.

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 that are used in 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 end by the conclusion and the future work in Section 7.

2. Related work

Schneider [25] 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 [1,18,19] 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 [19]. 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 [9] 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 onto action sets and (2) the mapping of action sequences onto the Java execution stack contents.

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 over Σ is denoted by Σ^ω, and $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$ denotes the set of all (finite or infinite) sequences over Σ. The empty sequence is denoted by ϵ. A sequence counting the actions $a_1, a_2, \ldots, a_n$ in this order is denoted by $a_1 a_2 \ldots a_n$. We use "a" to differentiate between the action a and the sequence counting only the action 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 $|\sigma|$ the length of a sequence σ. The set $\Sigma_k = \{\sigma \in \Sigma^* : |\sigma| = k\}$ denotes the set of all possible sequences of length k where k is a positive integer. For some positive integer k, $\Sigma_{\leqslant k} = \{\sigma \in \Sigma^* : |\sigma| \leqslant k\}$ denotes the set of all possible sequences of length less than or equal to k. The set $(\Sigma_{\leqslant k} \times \Sigma_{\leqslant k})_{\leqslant k} = \{(\sigma, \sigma') \in \Sigma_{\leqslant k} \times \Sigma_{\leqslant k} : |\sigma\sigma'| \leqslant k\}$ 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 $Fact_k(\sigma) = \{\sigma' \in \Sigma_k | \exists\, \sigma'' \in \Sigma^*. \exists \sigma''' \in \Sigma^\infty : \sigma = \sigma''\sigma'\sigma'''\}$. The sets $Pref_{\leqslant k}$ and $Pref_k$ are defined, respectively, by $\{\sigma' \in Pref(\sigma) : |\sigma'| \leqslant k\}$ and $\{\sigma' \in Pref(\sigma) : |\sigma'| = k\}$. Similarly, $Suf_{\leqslant k}(\sigma) = \{\sigma' \in Suf(\sigma) : |\sigma'| \leqslant k)\}$ and $Suf_k(\sigma) = \{\sigma' \in Suf(\sigma) : |\sigma'| = k)\}$.

We need also some definitions concerning security policies. A security policy P defines executions that are not acceptable according to some security standpoint. A security policy P is a *property* if there exists a predicate $\hat{P}$ over individual executions satisfying $\forall \sigma \in \Sigma^\infty.\ \sigma \in P \Leftrightarrow \hat{P}(\sigma)$ where Σ is the set of possible actions. Therefore, a security property P can be defined as a set of sequences such that $P \subseteq \Sigma^\infty$. Accordingly, a sequence σ satisfies a security property P if and only if $\sigma \in P$. 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

Execution monitors (EM) are enforcement mechanisms that control the execution of untrusted programs. They launch an intervention procedure when a controlled program is about to violate the policy being enforced. We denote by *conventional execution monitors* (CEM) those monitors for which the intervention procedure consists simply in halting the execution. We denote by *rewriter-based execution monitors* (RWEM) those monitors for which the intervention procedure is more powerful and may consist in inserting actions on behalf of the program or suppress potentially dangerous actions. A policy that can be enforced by an execution monitor is called EM-enforceable.

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

4.1. Security automata

According to Schneider [25], any CEM-enforceable security policy is prefix-closed. Access control [17], bounded availability [11], Chinese Wall [6], and One-Out-Of-k authorization [8] are examples of CEM-enforceable policies. Any CEM-enforceable policy can be specified by a *security automaton.*

In this characterization, a monitor can intervene only by halting the program execution. According to this definition of EM-enforcement, Schneider [25] observes that every EM-enforceable security policy P is a prefix-closed property. An EM-enforceable policy is specified by a *security automaton.*

Definition 4.1 (*Security automaton*)**.** A Security automaton (SA) [25] 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.
- $\delta : (Q \times \Sigma) \to Q$ 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 $q \overset{a}{\rightarrow} q'$ where $q' = \delta(q, a)$. We denote by $\delta^*(\sigma)$ the last state reached by the path recognizing a finite sequence σ. Let A_P denote the security policy specified by a *SA 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

In addition to halting the execution of the controlled program, RWEM can modify the program actions either by suppressing or inserting actions. A rewriter-based execution monitor can be specified by an edit automaton.

Definition 4.2 (*Edit automaton*)**.** An edit automaton (EA) is defined by a quadruple $\langle \Sigma, Q, q_0, \delta \rangle$ where:

- Σ is the set of finite or countably infinite input actions.
- Q is the set of finite or countably infinite automaton states.
- $q_0 \in Q$ is the initial state.
- $\delta : (Q \times \Sigma) \rightarrow (Q \times \Sigma^\infty)$ is the (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 a sequence τ to edit. The edited sequence τ specifies the intervention action to take by the execution monitor in order to enforce the property: (1) If $\tau =$ "a" then the input action a is to be accepted, (2) if $\tau \neq \epsilon \wedge \tau \neq$ "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 that the monitor can take in order 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. A valid path is a (finite or infinite) sequence of transition steps of the form $q_0 \underset{[\tau_1]}{\overset{\sigma[1]}{\rightarrow}} q_1 \underset{[\tau_2]}{\overset{\sigma[2]}{\rightarrow}} \ldots q_{n-1} \underset{[\tau_n]}{\overset{\sigma[n]}{\rightarrow}} q_n \ldots$ where $\forall 1 < i < |\sigma| . \delta(q_{i-1}, \sigma[i]) = \langle q_i, \tau_i \rangle$. Therefore, the sequence edited by the edit automaton A while reading the input sequence σ is $\tau_1\tau_2 \ldots \tau_n \ldots$ and it is denoted by $A(\sigma)$. 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*: For any input sequence σ, the sequence $A(\sigma)$ edited by an edit automaton A must satisfy the property P enforced by A, i.e., $A(\sigma) \in P$.
2. *Transparency*: The semantics of any execution satisfying the property must be preserved with respect to some equivalence relation.

Edit automata 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 4.3 (*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) $\sigma \in P \Rightarrow A(\sigma) \cong \sigma$.

[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.

It has been proved [1,19] that execution monitors that are specified by edit automata and acting as effective$_=$ enforcers[2] have more enforcement power than execution monitors that are specified by security automata. Indeed, an execution monitor acting as effective$_=$ enforcer 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 [1,18,19]. *Renewal properties* is identified as a lower bound of properties that are effectively$_=$ enforceable by edit automata.

Definition 4.4 (*Renewal property*)**.** A property P over Σ^∞ is a renewal property if and only if it satisfies one of the two following conditions:

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

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

Proposition 4.5. *[19] A property P over Σ^∞ is effectively$_=$ enforceable by edit automata if P is a renewal property and $\epsilon \in P$.*

Proof. Proposition 4.5 corresponds to Theorem 8 in [19]. The reason behind presenting a detailed proof of this proposition is twofold: (1) adapting the proof provided in [19] to Definition 4.2 and (2) explaining the ideas behind automata construction since they will be reused in many 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.
- $\delta : (Q \times \Sigma) \to (Q \times \Sigma^\infty)$ 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. We have the two following cases:

a. 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.

b. 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 valid 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 $Pref(\sigma')$. After editing the longest valid prefix σ' and reaching some state $q_{\sigma'}$, the automaton dos not accept any extension since the transition function is not defined for any input action from the state $q_{\sigma'}$. □

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

4.3. Fong's characterization

Fong [9] has proposed an information-based approach characterizing EM-enforceable security policies. The proposed characterization is based on the information about the execution history that is available to 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 4.6 (*Abstraction function[9]*)**.**
Let S be a finite or countably infinite set of abstract states and let α be any function such that $\alpha : \Sigma^* \to S$. 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 4.7 (α-SA*[9]*)**.** Let $\alpha : \Sigma^* \to S$ 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 the information left behind by the abstraction function [9]. 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 executed. The formal definition of shallow history automata is the following.

Definition 4.8 (*Shallow history automaton[9]*)**.** A shallow history automaton is a security automaton of the form $\langle \Sigma, 2^{\Sigma}, \emptyset, \delta \rangle$ where:

- Σ is the set of finite or countably infinite input symbols.
- 2^{Σ} is the set of finite or countably infinite automaton states. Each state represents a shallow history.
- $\emptyset$ is the initial (shallow history) state.
- $\delta : 2^{\Sigma} \times \Sigma \to 2^{\Sigma}$ 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 5.1 (*Bounded security automaton*)**.** A BSA of bound k (k-BSA) is a SA $\langle \Sigma, Q = \Sigma_{\leqslant k}, q_0, \delta \rangle$ 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).
- $\delta : (Q \times \Sigma) \to Q$ 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 $\beta : \Sigma_{\leqslant k+1} \to \Sigma_{\leqslant k}$ 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 5.2. *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 $A = \langle \Sigma, Q = \Sigma_{\leqslant k}, q_0, \delta \rangle$. The k'-BSA enforcing P is $A' = \langle \Sigma, Q' = \Sigma_{\leqslant k'}, q_0, \delta' \rangle$ where $\delta' : (\Sigma_{\leqslant k'} \times \Sigma) \to \Sigma_{\leqslant k'}$ is defined such that $\forall q \in \Sigma_{\leqslant k'}.\forall a \in \Sigma.\delta'(q, a) = \delta(q, a)$ 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 $P = \{Pref(a_1 \ldots a_{k'+1})\}$ where the set of input actions is defined by $\Sigma = \{a_1, \ldots, a_{k'+1}\}$. Indeed, to recognize the sequence $a_1 \ldots a_{k'+1}$ we need to save the history of the last k' actions which is not possible by any k-BSA since $k < k'$. □

5.2. Enforcing bounded availability properties using BSA

Bounded availability properties specify that any acquired resource must be released by some fixed point later in the execution. According to [25], a bounded availability property is EM-enforceable if it is specified such that any resource cannot be acquired more than some MWT (maximum waiting time) execution steps without being released. Enforcing such properties protects systems from denial of service attacks [11]. One can easily prove that any k bounded availability property can be enforced by some k-BSA. Fig. 1 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 two steps. The set of resources is $\{A, B\}$. Actions a and b represent acquiring resource A and B, respectively, and actions $\bar{a}$ and $\bar{b}$ 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 property, each execution must satisfy the following rules:

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

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

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

$$\begin{array}{lllll} \alpha(a) = a & \alpha(a\tau) = a\tau & \alpha(a\bar{a}) = \epsilon & \alpha(ab) = ab & \alpha(a\tau\bar{a}) = \epsilon \\ \alpha(b) = b & \alpha(b\tau) = b\tau & \alpha(b\bar{b}) = \epsilon & \alpha(ba) = ba & \alpha(b\tau\bar{b}) = \epsilon \end{array}$$

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

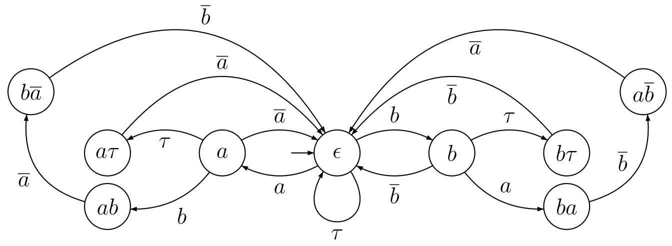


Fig. 1. A bounded security automaton enforcing the *Two-BA* property.

The three following results explain the connection between BSA and Fong's α-*SA* and SHA (the definitions of abstraction functions, α-*SA*, and SHA have been presented in 4.6, 4.7, and 4.8, respectively).

Proposition 5.3. *For any k-BSA* $A = \langle \Sigma, Q = \Sigma_{\leqslant k}, q_0, \delta\rangle$, *there exists an* α-*SA enforcing* A_{P_f}.

Proof. The α-*SA* enforcing A_{P_f} is defined by the security automaton $A' = \langle \Sigma, Q' = Q \cup \{bad\}, \alpha(\epsilon), \delta'\rangle$ 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:
  $\forall \sigma \in \Sigma^*.\alpha(\sigma) = \begin{cases} \delta^*(\sigma) \text{ if } \sigma \in A_{P_f} \\ \text{bad if } \sigma \notin A_{P_f} \end{cases}$

  The abstraction function α satisfies the compatibility property (3), i.e., for any action $a \in \Sigma$ and any two finite sequences $\sigma, \sigma' \in \Sigma^*$, if $\alpha(\sigma) = \alpha(\sigma') = h$ then $\alpha(\sigma a) = \alpha(\sigma' a)$. We have the two following cases:
  - If $h = bad$ then $\sigma, \sigma' \notin A_{P_f}$ and consequently any extension of σ or σ' is not in A_{P_f}. Therefore, $\sigma a, \sigma' a \notin A_{P_f}$ 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 $h' \in Q$ 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 $h' \in Q$ 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 5.4. *Let* $A = \langle \Sigma, Q, \alpha(\epsilon), \delta\}\rangle$ *be an* α-*SA enforcing a property P. The property P is BSA-enforceable if*:

$$\exists k' \in \mathbf{N}^+ . |Q| \leqslant |\Sigma_{\leqslant k'}| \tag{4}$$

Proof. We prove this result by constructing the k-BSA enforcing the property P. If 4 is satisfied then we have:

$$\exists \beta' : Q \rightarrow \Sigma_{\leqslant k'} . \forall q, q' \in Q . q \neq q' \Rightarrow \beta'(q) \neq \beta'(q') \tag{5}$$

Let k be the smallest positive integer satisfying 4, and let β be any mapping function satisfying 5. Note that if the set of abstract states Q is finite, then the existence of k and β is guaranteed and $k \leqslant |Q|$.

The k-BSA enforcing P is defined by the *SA* $A' = \langle \Sigma, Q' = \Sigma_{\leqslant k}, \beta(\alpha(\epsilon)), \delta'\rangle$ where the transition function $\delta' : (Q' \times \Sigma) \rightarrow Q'$ is defined such that for any abstract state $q \in Q$ 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 5.5. *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 5.4. Let $A = \langle \Sigma, 2^{\Sigma}, \emptyset, \delta \rangle$ be the shallow history automaton enforcing the property P. If Σ is finite then the set 2^{Σ}, which is the set of all states of A, is finite also and Condition 4 is satisfied. Indeed, the smallest positive integer satisfying Condition 4 is $k = |\Sigma|$ and the mapping function can be $Act \quad : \Sigma_{\leqslant k} \rightarrow 2^{\Sigma}$ which is the function that returns, for each sequence σ, the set of actions that are present in σ. Therefore, by Proposition 5.4, there exists a BSA enforcing P. The k-BSA enforcing P is defined by $\langle \Sigma, \Sigma_{\leqslant k}, \epsilon, \delta' \rangle$ where the transition function δ' is defined by: $\forall \sigma \in \Sigma_{\leqslant k}. \forall a \in \Sigma$.

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

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. □

Corollary 5.6. *Let $A = \langle \Sigma, 2^{\Sigma}, \emptyset, \delta \rangle$ be a SHA enforcing a property P. The property P is BSA-enforceable if:*

$$\exists k' \in \mathbf{N}^{+}. \forall q \in Q'. |q| \leqslant k'. \tag{6}$$

where $Q' \subset Q$ is the set of states that are actually used to define the sequences of P.

Proof. We prove this result by constructing the k-BSA enforcing the property P. Let k be the smallest positive integer satisfying 6. The k-BSA enforcing P is defined by the *SA* $A' = \langle \Sigma, \Sigma_{\leqslant k}, \epsilon, \delta' \rangle$ where the transition function δ' is defined by: $\forall \sigma \in \Sigma_{\leqslant k}. \forall a \in \Sigma$:

$$\delta'(\sigma, a) = \begin{cases} \sigma \text{ if } a \in Act \quad (\sigma) & (a) \\ \sigma a \text{ if } a \notin Act \quad (\sigma) \wedge \delta(Act \quad (\sigma), a) = Act \quad (\sigma) \cup \{a\} & (b) \\ \text{Undefined, otherwise.} & (c) \end{cases} \qquad \square$$

5.2.1. K-BSA vs α-SA

Even if, by Proposition 5.3, any k-BSA can be viewed as an α-*SA*, k-BSA are more precise than α-*SA* for specifying properties to be enforced under memory-limitation constraints. Indeed, while α-*SA* provides the exact information needed to enforce a policy by an execution monitor, k-BSA provides an estimation of the memory size used by the execution monitor to track that information.

If Condition 4 (Proposition 5.4) is not satisfied, then the amount of information left behind by the abstraction function α is too broad to be tracked by a bounded history. This implies that enforcing the policy specified by such α-*SA* may require tracking the full execution history. Viewed in this light, Condition 4 can be used as a selection criteria on security policies to enforce on memory-limited systems. Indeed, a security policy cannot be enforced under memory-limitation constraints if there is no abstraction function considerably reducing the amount of information needed to enforce that policy.

Finally, by Corollary 5.5, 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. However, requiring in Corollary 5.5, that the input actions set must be a finite set is too restrictive. Although the state set of a SHA is infinite when the input actions set is infinite, the number of states, actually used to define the sequences of a SHA-enforceable policy is not necessary infinite. Therefore Corollary 5.6 specifies ,when the input actions set is infinite, condition to satisfy by a SHA-enforceable property in order to be BSA-enforceable.

5.3. Enforcing SHA-enforceable properties using BSA

Corollary 5.5 allows the identification of any SHA-enforceable property (when the actions set Σ is finite) as BHA-enforceable. In the sequel, we show how some SHA-enforceable properties can be enforced using BSA.[3]

[3] These properties were provided by Fong in [9].

5.3.1. 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 [9] and by Corollary 5.5, it is enforceable by some k-BSA if the subject set S and the object set O are finite and $|S \times O| = k$.

5.3.2. 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 $l(s) \geqslant l(o)$.
- $exec(s,s')$ is allowed if and only if $l(s) \geqslant l(s')$.

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 [9] and by consequence it is enforceable by some k-BSA if the set Σ is finite and $k = |\Sigma|$.

5.3.3. One-Out-Of-k Authorization policy

The One-Out-Of-k Authorization policy [8] 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 [8] 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 [9] and consequently is enforceable by some k-BSA if the set of all possible actions Σ is finite such that $k = |\Sigma|$.

5.3.4. Assured pipelines policy

Assured pipelines [31,5] 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 [9]:

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

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

5.4. 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 [19].

Definition 5.7 (*Bounded edit automaton*)**.** A BEA of bound k (k-BEA) is an EA $\langle \Sigma, Q = (\Sigma_{\leqslant k} \times \Sigma_{\leqslant k})_{\leqslant k}, q_0, \delta \rangle$ 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 $\sigma_{Acc}\sigma_{Sup} \in \Sigma_{\leqslant k}$.
- 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.
- $\delta : (Q \times \Sigma) \to (Q \times \Sigma^\infty)$ 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 $\beta : \Sigma_{\leqslant k+1} \to \Sigma_{\leqslant k}$ 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 $\alpha : \Sigma_{\leqslant k} \to (\Sigma_{\leqslant k} \times \Sigma_{\leqslant k})_{\leqslant k}$ 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 5.8. *For any two positive integers k and k' such that $k < k'$, we have $EM_{kEA} \subset EM_{k'EA}$.*

Proof. This theorem can be easily proved by following the same intuition used to prove Theorem 5.2.

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 $A = \langle \Sigma, Q = (\Sigma_{\leqslant k} \times \Sigma_{\leqslant k})_{\leqslant k}, q_0, \delta \rangle$. The k'-BEA enforcing P is $A' = \langle \Sigma, Q = (\Sigma_{\leqslant k'} \times \Sigma_{\leqslant k'})_{\leqslant k'}, q_0, \delta' \rangle$ where the transition function $\delta' : ((\Sigma_{\leqslant k'} \times \Sigma_{\leqslant k'})_{\leqslant k'} \times \Sigma) \to (\Sigma_{\leqslant k'} \times \Sigma_{\leqslant k'})_{\leqslant k'}$ is defined such that:
$\forall q \in (\Sigma_{\leqslant k'} \times \Sigma_{\leqslant k'})_{\leqslant k'} . \forall a \in \Sigma . \delta'(q, a) = \delta(q, a)$ 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 $P = \{a_1 \ldots a_{k'+1}\}$ where the set of input actions is defined by $\Sigma = \{a_1, \ldots, a_{k'+1}\}$. Indeed, to recognize the sequence $a_1 \ldots a_{k'+1}$ we need to save the history of the last k' actions which is not possible by any k-BEA since $k < k'$. □

5.5. Enforcing transaction-based properties using BEA

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 [22] 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-based 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

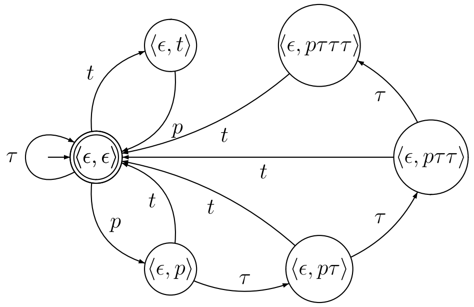


Fig. 2. A bounded edit automaton enforcing a transaction-based property.

with bounded history automata, constraints have to be imposed on the size of elements of T. Indeed, if $P = T^\infty$ is a transaction-based property then P is k-BEA-enforceable if and only if $T \subseteq \Sigma_{\leqslant k+1}$.

Fig. 2 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:

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

5.6. *Bounded history-based taxonomy of EM-enforceable policies*

Theorems 5.2 and 5.8 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 5.2, we get the following taxonomy: $EM_{0SA} \subset EM_{1SA} \subset EM_{2SA} \ldots \subset EM_{SA}$. The smallest class of this taxonomy is the class of properties that are enforceable by BSA having no space to save the execution history and the biggest class is the class of properties that are enforceable by security automata 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 5.8, we get the following taxonomy: $EM_{0EA} \subset EM_{1EA} \subset EM_{2EA} \ldots \subset EM_{EA}$. Note that for any positive integer k, we have $EM_{kSA} \subset EM_{kEA}$.

6. Bounded history automata and local testability

Locally testable (LT) properties [3] are identified as the class of properties recognizable by inspecting "local" information. These properties have been well studied in the literature, mainly, from the standpoints of language and semigroup theories [3,7,23,24,15]. Also, the problem of deciding whether a property is LT has been well investigated and many algorithms have been proposed [13,14,28]. LT properties have practical importance since they are characterized by a low memory demand for their verification. In this section, we investigate the connection between LT properties and BHA-enforceable properties. More precisely, we (1) identify the conditions under which a LT property is BHA-enforceable and (2) show how to use LT properties decision algorithm to identity BHA-enforceable properties.

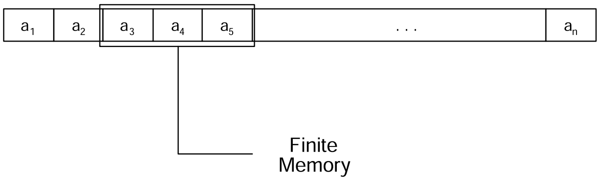


Fig. 3. A scanner.

6.1. Locally testable properties

Locally testable properties are properties recognizable by *scanners* (Fig. 3); 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 accepts or to rejects the input sequence.

In the sequel, we present the different classes of LT properties that we can found in the literature.[4] The main classes of locally testable properties are *strictly locally testable*, *prefix testable*, *suffix testable*, *prefix–suffix testable*, and *strongly locally testable*. It is important to note that the definitions of LT properties over infinite sequences are presented in such a form that LT properties can easily be viewed as renewal properties (Definition 4.4). This will facilitate the proofs of theorems identifying BEA-enforceable LT properties (Section 6.3). For each LT class, we define the conditions (if any) that a property must satisfy to be prefix-closed. This will help to identify BSA-enforceable LT properties.[5]

We start by the class of strictly LT properties since the other LT properties classes can be viewed as subclasses of it.

Definition 6.1 (*Strictly locally testable property*).

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

$$\forall \sigma \in \Sigma^*.\sigma \in L \Leftrightarrow (\sigma \in X) \vee ((\sigma[..k-1] \in P) \wedge (\sigma[|\sigma|-k+2..] \in S) \wedge (Fact_k(\sigma) \subseteq F)) \tag{7}$$

$$\forall \sigma \in \Sigma^\omega.\sigma \in L \Leftrightarrow \forall \sigma' \in Pref(\sigma).\exists \sigma'' \in Pref(\sigma).\sigma' \in Pref(\sigma'') \wedge \sigma'' \in L \tag{8}$$

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

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

According to this definition, the set of all sequences of a strictly LT 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 $\sigma \in \Sigma_{\leqslant k-1}$ 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:

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

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.

[4] The definitions of LT properties provided in this paper cover both finite and infinite sequences. In the literature, LT properties over finite sequences and LT properties over infinite sequences are treated separately [7,23].

[5] For the sake of paper readability, the proofs related to prefix-closed LT properties are presented in Appendix A.

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 LL can also be written as $LL \ = \{aba\}^* \cup \{aa, bb\}$.

Since the definition of a strictly LT 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 in F while there is no sequence of L that can have f as a factor. For example, if for the property LL defined above, the set F is defined such that $F = \{aba, baa, aab, bbb\}$, then there is no sequence σ of LL counting bbb as a factor.

For some results of this section we need the exact definition of the factors actually used to construct the sequences of a strictly LT property. Let L be a strictly k-LT property defined by the sets P, S, F, and X. We denote by F_R, the set of all factors actually used to construct the sequences of L. The set F_R is 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 6.2. *Let k be any positive integer, and let $F \subseteq \Sigma_k$, $P, S \subseteq \Sigma_{k-1}$, and $X \subseteq \Sigma_{\leqslant k-1}$ be the sets used to define a strictly 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. Please see Appendix A for the proof. □

We start by *prefix-testable* properties that are recognizable by inspecting only prefixes of limited size.

Definition 6.3 (*Prefix-testable property*). 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 $X \subseteq \Sigma_{\leqslant k-1}$ such that the elements of L are defined by the following rule:

$$\forall \sigma \in \Sigma^\infty . \sigma \in L \Leftrightarrow (\sigma \in X) \vee (\sigma[..k] \in P). \tag{9}$$

A property L of Σ^∞ is prefix testable (PT) 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 6.4. *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 $X \subseteq \Sigma_{\leqslant k-1}$. The property L is prefix-closed if and only if $X \cup P$ is prefix-closed.*

Proof. Please see Appendix A for the proof. □

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

Definition 6.5 (*Suffix testable property*). 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 $X \subseteq \Sigma_{\leqslant k-1}$ such that the elements of L are defined by the two following rules:

$$\forall \sigma \in \Sigma^* . \sigma \in L \Leftrightarrow (\sigma \in X) \vee (\sigma[|\sigma| - k + 1..] \in S) \tag{10}$$

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

where $\sigma[|\sigma| - k + 1..]$ is the suffix of σ of length k.
A property L of Σ^∞ is suffix testable (ST) 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 6.6 (*Prefix–suffix testable property*). 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 $X \subseteq \Sigma_{\leqslant k-1}$ such that the elements of L are defined by the two following rules:

$$\forall \sigma \in \Sigma^* .\ \sigma \in L \Leftrightarrow \sigma \in X \vee (\sigma[..k] \in P \wedge \sigma[|\sigma - k + 1..] \in S) \tag{12}$$

$$\forall \sigma \in \Sigma^{\omega}.\ \sigma \in L \Leftrightarrow \forall \sigma' \in Pref(\sigma).\ \exists \sigma'' \in Pref(\sigma).\sigma' \in Pref(\sigma'') \wedge \sigma'' \in L \tag{13}$$

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

A property L of Σ^{∞} is prefix–suffix testable (PST) if it is k-prefix–suffix testable for some integer k.

According to this definition, the set of all sequences of a PST 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 $L \cap \Sigma^{\omega} = P\Sigma^{\omega} \cap (\Sigma^* S)^{\omega}$.

The following proposition states that ST and PST properties are not prefix-closed. This result will be used in 6.2 to deduce that these property classes are not BSA-enforceable.

Proposition 6.7. *Let k be any positive integer, and let P, S, X be three sets such that $P, S \subseteq \Sigma_k$, and $X \subseteq \Sigma_{\leqslant k-1}$. If L is a k-ST property defined by the two sets S and X or a k-PST property defined by the sets P, S, and X, then L is not prefix-closed.*

Proof. Please see Appendix A for the proof. □

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

Definition 6.8 (*Strongly locally testable property*)**.** Let k be a positive integer. A property L of Σ^{∞} is k-strongly locally testable (k-SLT) if there exists a set $F \subseteq \Sigma_k$ such that the elements of L are defined by the following rule:

$$\forall \sigma \in \Sigma^{\infty}.\sigma \in L \Leftrightarrow \forall \sigma' \in Pref(\sigma).Fact_k(\sigma') \subseteq F \tag{14}$$

A property L of Σ^{∞} is strongly locally testable (SLT) if it is k-strongly locally testable for some integer k.

Let $\bar{F} = \Sigma_k \setminus F$ be the complement of F in Σ_k. According to Definition 6.8, the set of all sequences of a SLT property L is defined by $L = \Sigma^{\infty} \setminus (\Sigma^* \bar{F} \Sigma^{\infty})$. The set of all finite sequences of L is defined by the set $(L \cap \Sigma^*) = \Sigma^* \setminus (\Sigma^* \bar{F} \Sigma^*)$. Similarly, the set of all infinite sequences of L is defined by the set $L \cap \Sigma^{\omega} = \Sigma^{\omega} \setminus (\Sigma^* \bar{F} \Sigma^{\omega})$.

Proposition 6.9. *Let k be any positive integer, and let $F \subseteq \Sigma_k$. If L is a k-SLT property defined by the set F, then L is prefix-closed.*

Proof. Please see Appendix A for the proof. □

6.2. BSA-enforceable locally testable properties

In what follows, we study the problem of deciding whether a LT property is BSA-enforceable. The results of this section will help to check whether a LT property is enforceable by a memory-constrained CEM when the CEM is characterized by some k-BSA. Since BSA is a class of security automata, only prefix-closed LT properties are BSA-enforceable. Based on this rule, the following theorems identify BSA-enforceable LT properties and LT properties that are not BSA-enforceable. The proofs[6] related to those BSA-enforceable LT properties detail the construction of a BSA enforcing a given LT property.

Theorem 6.10. *Let k be any positive integer. Any prefix-closed strictly k-LT testable property L is enforceable by some k-BSA.*

Proof. From Proposition 6.2, 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 $\langle \Sigma, Q = \Sigma_{\leqslant k}, k, \epsilon, \delta \rangle$ where the transition function δ is defined by:

[6] For the sake of paper readability, only the theorem proof related to BSA-enforceable strictly LT properties is detailed in this section. The other proofs are presented in Appendix A and can be deduced from the proof detailed here.

$$\forall \sigma \in Q. \forall a \in \Sigma.\ \delta(\sigma, a) = \begin{cases} \sigma a \text{ if } \sigma a \in X \cup F_{initial} & (a) \\ \sigma[2..]a \text{ if } (\sigma \in F_R \wedge \sigma[2..]a \in F_R) & (b) \\ \text{Undefined, otherwise.} & (c) \end{cases}$$ 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 $|\sigma| \leqslant k$, *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 $\Sigma_{\leqslant k} \setminus (X \cup F_{initial})$ cannot 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 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 $\epsilon \xrightarrow{\sigma'[1]} \sigma'[1] \ldots \sigma'[k \quad - 1 \quad] \xrightarrow{\sigma'[k]} \sigma' \xrightarrow{\sigma''[1]} f_1 \xrightarrow{\sigma''[2]} f_2 \ldots f_{d-1} \xrightarrow{\sigma''[d]} f_d$ where $\sigma' \in F_{initial}$, $|\sigma''| = d$, $f_1 = \sigma'[2..]\sigma''[1]$, $\forall\, 1 < i \leqslant d. f_i = f_{i-1}[2..]\sigma''[i]$, $\forall\, 1 \leqslant i \leqslant d. f_i \in F_R$, and $f_d \in F_{terminal}$. It is obvious that no sequence outside L can be recognized by A. □

Given a strictly k-LT property L, one can (1) use Proposition 6.2 to verify whether L is prefix-closed and if so, (2) use the automata construction technique detailed in the proof of Theorem 6.10 to construct the k-BSA enforcing L.

Theorem 6.11. *Let k be any positive integer. Any prefix-closed k-PT property L is enforceable by some k-BSA.*

Proof. The proof of this theorem is similar to the proof of Theorem 6.10. It is presented in Appendix A. □

Similarly, Given a k-LT property L, one can (1) use Proposition 6.4 to verify whether L is prefix-closed and if so, (2) use the automata construction technique detailed in the proof of Theorem 6.11 to construct the k-BSA enforcing L.

Theorem 6.12. *Let k be any positive integer. Any k-SLT property L is enforceable by some k-BSA.*

Proof. The proof of this theorem is similar to the proof of Theorem 6.10. It is presented in Appendix A. □

Given a k-SLT property L, one can directly use the automata construction technique detailed in the proof of Theorem 6.12 to construct the k-BSA enforcing L, since L is prefix-closed according to Proposition 6.9.

Theorem 6.13. *ST properties and PST properties are not BSA-enforceable.*

Proof. The proof can be immediately deduced from Proposition 6.7. Indeed, ST properties and PST properties are not prefix-closed and consequently they are not enforceable by security automata. □

Given a ST or a PST property L, one can deduce from Theorem 6.13, that L is not BSA-enforceable and consequently not enforceable by memory-constrained CEMs.

6.3. BEA-enforceable locally testable properties

In what follows, we study the problem of deciding whether a LT property is BEA-enforceable. Based on the results of this section, one can check whether a LT property is enforceable by a memory-constrained RWEM when the RWEM is characterized by some k-BEA. The following theorems define conditions (if any) under which a LT property is BEA-enforceable. For any BEA-enforceable LT property L, the theorem proof[7] detail how to construct a BEA enforcing L.

The following definition is needed for the first theorem.

Definition 6.14.

Let L be a strictly k-LT property defined using the sets X, P, S and F. Then for any factor $f \in F_{initial}$, we define x_f as the longest element of the set $X \cap Pref(f)$.

Intuitively, *Sup*(f) represents the longest sequence edited by the edit automaton after reading the factor f.

Theorem 6.15. *Let k be any positive integer. A strictly k-LT property L is enforceable by some k-BEA if and only if L satisfies the followings*:

$$\forall f_i \in F_{initial}.\forall f \in F \setminus F_{terminal}.Fact(x_{f_i}f) \subseteq F \Rightarrow Fact(x_{f_i}f) \cap F_{terminal} \neq \emptyset \tag{15}$$

$$\forall f_t \in F_{terminal}.\forall f \in F \setminus F_{terminal}.Fact(f_t f) \subseteq F \Rightarrow Fact(f_t f) \cap F_{terminal} \neq \emptyset \tag{16}$$

Proof. We prove this result by constructing the BEA enforcing L. Let k be any positive integer and let $P, S \subseteq \Sigma_{k-1}$, $X \subseteq \Sigma_{\leqslant k-1}$ and $F \subseteq \Sigma_k$ be the sets used to define the strictly k-LT property L over Σ^∞. The BEA enforcing L is defined by the $EA\ A = \langle \Sigma, Q = (\Sigma_{\leqslant k} \times \Sigma_{\leqslant k})_{\leqslant k}, \langle \epsilon, \epsilon \rangle, \delta \rangle$ of bound k where the partial transition function δ is defined such that $\forall \langle \sigma_{Acc}, \sigma_{Sup} \rangle \in Q.\forall a \in \Sigma$.

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

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

(c) Undefined otherwise.

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

- Case $|\sigma| \leqslant k$, 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 $\langle \epsilon, \epsilon \rangle \underset{[\tau_1]}{\overset{\sigma[1]}{\rightarrow}} \langle \sigma_{1Acc}, \sigma_{1Sup} \rangle \underset{[\tau_2]}{\overset{\sigma[2]}{\rightarrow}} \dots \underset{[\tau_{m-1}]}{\overset{\sigma[m-1]}{\rightarrow}} \langle \sigma_{m-1Acc}, \sigma_{m-1Sup} \rangle \underset{[\tau_m]}{\overset{\sigma[m]}{\rightarrow}} \langle \sigma, \epsilon \rangle$ where $|\sigma| = m$ and $\forall i. 1 < i < m. \tau_i = \delta(\langle \sigma_{i-1Acc}, \sigma_{i-1Sup} \rangle, \sigma[i])$, and $\forall i. 1 \leqslant i < m.\ \sigma_{iAcc}\sigma_{iSup} = \sigma[..i] \wedge (\sigma[..i] \in X \cup (F_{initial} \cap F_{terminal}) \Rightarrow \sigma_{iSup} = \epsilon)$.
- 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 $\langle \epsilon, \epsilon \rangle \rightarrow^{\sigma'[1]}_{[\tau'_1]} \langle \sigma'_{1Acc}, \sigma'_{1Sup} \rangle \underset{[\tau'_2]}{\overset{\sigma'[2]}{\rightarrow}} \dots \rightarrow^{\sigma[m-1]}_{[\tau'_{m-1}]} \langle \sigma'_{m-1Acc}, \sigma'_{m-1Sup} \rangle$ $\underset{[\tau'_m]}{\overset{\sigma'[m]}{\rightarrow}} \langle \sigma'_{Acc}, \sigma'_{Sup} \rangle \rightarrow^{\sigma''[1]}_{[\tau''_1]} \longrightarrow \langle f_{1Acc}, f_{1Sup} \rangle \underset{[\tau''_2]}{\overset{\sigma''[2]}{\rightarrow}} \dots$ $\underset{[\tau''_{d-1}]}{\overset{\sigma''[d-1]}{\rightarrow}} \langle f_{d-1Acc}, f_{d-1Sup} \rangle \underset{[\tau''_d]}{\overset{\sigma''[d]}{\rightarrow}} \longrightarrow \langle f_d, \epsilon \rangle$ where:
  - $|\sigma'| = k$ and $|\sigma''| = d$.

[7] For the sake of paper readability, only the theorem proof related to BEA-enforceable strictly LT properties is detailed in this section. The other proofs that can be deduced from that proof are presented in Appendix A.

- $\forall i.1 < i \leqslant k.\tau'_i = \delta(\langle \sigma'_{i-1_{Acc}}, \sigma'_{i-1_{Sup}} \rangle, \sigma'[i])$
- $\forall i.1 \leqslant i \leqslant k.\ (\sigma'[..i] \in X \cup (F_{initial} \cap F_{terminal}) \Rightarrow \sigma'_{iSup} = \epsilon) \wedge \sigma'_{iAcc}\sigma'_{iSup} = \sigma'[..i]$
- $f_{1Acc}f_{1Sup} = \sigma'[2..]\sigma''[1]$.
- $\forall\, 1 \leqslant i \leqslant d.(f_{iAcc}f_{iSup} \in F_{terminal} \Rightarrow (f_{iAcc} \in F_{terminal} \wedge f_{iSup} = \epsilon) \wedge f_{iAcc}f_{iSup} \in F$.

By Definition 6.1, a strictly LT 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 6.1, any invalid prefix σ' can be extended to some valid prefix σ'' which can be recognized by a finite path as described above. □

Given a strictly k-LT property L, one (1) can check if L satisfies conditions (15) and (16) and if so, (2) use the automata construction technique detailed in the proof of Theorem 6.15 to construct the k-BEA enforcing L. Theorem 6.15 proves that BEA enforce more strictly LT properties than BSA since it does not require a strictly LT property to be prefix-closed. However BEA cannot enforce any strictly LT property since they cannot suppress more than k actions without reediting any suppressed action. Therefore there is no k-BEA that can enforce a strictly k LT property accepting some sequence σ for which there exists some subsequence σ' such that there is no σ' factor ending by a valid suffix.

Theorem 6.16. *Let k be any positive integer. Any k-PT property is enforceable by some k-BEA.*

Proof. The proof is presented in Appendix A. The construction of the k-BEA enforcing some k-PT property can be deduced from the proof of Theorem 6.15. □

Given a k-PT property L, one can use the automata construction technique detailed in the proof of Theorem 6.16 to construct the k-BEA enforcing L.

Theorem 6.17. *Let k be any positive integer. Any k-SLT property is enforceable by some k-BEA.*

Proof. The proof is presented in Appendix A. As for the previous theorem proof, the construction of the k-BEA enforcing some k-SLT property can be deduced from the proof of Theorem 6.15. □

Similarly, given a k-SLT property L, one can use the automata construction technique detailed in the proof of Theorem 6.17 to construct the k-BEA enforcing L.

Theorem 6.18. *If $|\Sigma| > 1$ then any ST property is not BEA-enforceable.*

Proof. We have to prove that if $|\Sigma| > 1$ then for any ST property, there is no BEA enforcing it. We proceed by contradiction. For some positive integer k, let P be any k-ST property defined by the two sets $S \subseteq \Sigma_k$ and $X \subseteq \Sigma_{\leqslant k-1}$. Let us suppose that there exists a BEA $A = \langle \Sigma, Q \subseteq (\Sigma_{\leqslant k'} \times \Sigma_{\leqslant k'})_{\leqslant k'}, q_0, \delta \rangle$ 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 $\sigma = \sigma'\sigma''s$ where (1) s is any suffix from S, (2) $\sigma' \in P$, (3) $Fact_k(\sigma'') \cap S = \emptyset$, and (4) $|\sigma''| > k'$. Intuitively, in order to recognize σ, we need to suppress the entire subsequence σ'' and save 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 BEA enforcing P. The existence of the sequence σ'' is guaranteed by the fact that (a) $|\Sigma| > 1$ and (b) P is really specifying a suffix testable property. Indeed, if $\neg(|\Sigma| > 1)$, i.e., $|\Sigma| = 1$, e.g., $\Sigma = \{a\}$ then $P = (\{a\}^* a^k)^\infty \cup X = a^k a^\infty \cup X$ which is a prefix-testable property! In this case, the property P is studied as suffix testable while, in reality, it specifies a prefix-testable property. For the case $|\Sigma| > 1$ and P is really specifying a suffix testable property, the existence of σ'' is ensured. Indeed, if there is no sequence σ'' satisfying (3) and (4) then there exists some integer $k'' \leqslant k'$ and some set $R \subseteq \Sigma_{\leqslant k''}$ such that $P = (RS)^\infty \cup X$. Let Pr and F be two sets defined such that $Pr = Pref_{k-1}(\{\sigma s | \sigma \in R \wedge s \in S\})$ and $F = Fact_k(\{\sigma s | \sigma \in R \wedge s \in S\}) \cup Fact_k(\{s\sigma | \sigma \in R \wedge s \in S\})$. Then, the property P can be viewed as a strictly

k-locally testable property defined by the sets Pr, S, X, and F. Indeed $P = ((Pr\Sigma^{\infty} \cap (\Sigma^{*}S)^{\infty}) \setminus \Sigma^{*}\overline{F}\Sigma^{\infty}) \cup X$ where $\overline{F} = \Sigma^{k} \setminus F$. Thus the property P is studied as suffix testable property while in reality, it is specifying a strictly locally testable property which is in contradiction with (b). □

According to Theorem 6.18, BEA cannot enforce ST properties since they cannot suppress more than k actions without reediting any suppressed action. Indeed, in the general case ($|\Sigma| > 1$), for any positive k value, one can find some sequence $\sigma'\sigma''s$ where s is a valid suffix and $|\sigma''| > k$ such that there is no prefix ending by a valid suffix, i.e., a suffix satisfying the ST property to enforce. For such a sequence σ'', the k-BEA suppresses the first k actions before halting the execution since it cannot continue accepting input actions without reediting any suppressed action. However, halting the execution will prevent the valid sequence $\sigma'\sigma''s$ from being executed. If we cannot find the sequence σ'', then according to the proof of Theorem 6.18, the property is not really k-suffix testable since it can be viewed as strictly k-locally testable. In this case, studying its BEA-enforceability as strictly LT property using Theorem 6.15 is more interesting than studying its BEA-enforceability as ST property using Theorem 6.18.

If $|\Sigma| = 1$, e.g., $\Sigma = \{a\}$, then for any k-ST property L, we can construct a k-BEA enforcing it. The k-BEA suppresses the first k actions of the execution and reedit the suppressed subsection each time it identifies an element of X. After reading the first k actions, the k-BEA simply accepts any input action because at any time the sequence length is greater than k and we have a suffix of k actions which satisfies the property.[8] As one can see, such properties are not interesting since they can be viewed as prefix-testable properties, by taking the same set X and the set S as a set of prefixes.

Theorem 6.19. *If $|\Sigma| > 1$ then any PST property is not BEA-enforceable.*

Proof. The proof of this theorem is similar to the proof of Theorem 6.18. It is presented in Appendix A. □

Similarly, in the general case of $k > 1$ and according to Theorem 6.19, BEA cannot enforce PST properties since they cannot suppress more than k actions without reediting any suppressed action.

The following two propositions show that ST properties and PST properties can be enforced by unbounded edit automata.

Theorem 6.20. *Let k be any positive integer. Any k-ST property L is enforceable by some edit automaton.*

Proof. Please see Appendix A for the proof. □

Theorem 6.21. *Let k be any positive integer. Any k-PST property L is enforceable by some edit automaton.*

Proof. Please see Appendix A for the proof. □

Although ST and PST properties are not BEA-enforceable, Theorem 6.21 proves that these two LT properties classes can be enforced by unbounded EA. In fact, all LT properties classes presented in this paper are EA-enforceable since they are defined as renewal properties.

6.4. Locally testable EM-enforceable properties

In 6.2 and 6.3, we investigated BHA-enforceable local properties. According to the provided results, one can check whether a local property is BHA-enforceable. For any BHA-enforceable local property, we showed how to construct the BHA enforcing it.

However, in practice, security policies are rarely specified as local properties. Indeed, security policies are usually specified in formalisms easily translatable to finite automata. Fortunately, deciding whether a property is local has been well investigated and many deciding algorithms have been proposed [28,14,13,21]. Since the majority of existing algorithms take as input properties specified by deterministic finite automata, we investigate in the sequel the translation of SA and EA into deterministic finite automata.

[8] In this special case, the set S used to define any k-ST property consists of only one k-length sequence σ where each σ action is equal to a.

Because we are dealing with both finite and infinite sequences, Büchi automata seem to be the most suitable automata model for specifying EM-enforceable security policies.[9] First we recall the formal definition of Büchi automata and explain their property recognition mode.[10]

Definition 6.22 (*Büchi automata [24]*)**.** A Büchi automaton B 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 is presented in the following:

(1) A finite sequence σ such that $|\sigma| = n$ is recognizable by Büchi automaton B, 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 $\forall\, 0 \leqslant i \leqslant n.\ q_i \in Q. \forall\, 0 \leqslant i < n.\ \delta(q_i, \sigma[i+1]) = q_{i+1}$ and $q_n \in F$. Therefore, σ is recognizable by a finite path starting from the initial state q_0 and ending in a final state.
(2) An infinite sequence σ is recognizable by Büchi automaton B, 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 6.23. *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 final states. □

Definition 4.2 allows us to view EA characterizing effective$_=$-enforcers as sequence recognizers rather than sequence transformers. Although EA were introduced as sequence transformers, the main relevant contributions targeting EA-enforcement were demonstrated using EA acting as effective$_=$-enforcers. Indeed, in [19,18,1], an effective$_=$-enforcer is characterized by an EA 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 EA acting as effective$_=$-enforcer.

Proposition 6.24. *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' such that $A' = \langle \Sigma, \Sigma^* \times \Sigma^*, \langle \epsilon, \epsilon \rangle, F, \delta' \rangle$ 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: $\delta'(q, a) = \begin{cases} q' \text{ if } \delta(q, a) = (q', \tau). \\ \text{Undefined, if } \delta \text{ is not defined for the pair } (q, a). \end{cases}$ □

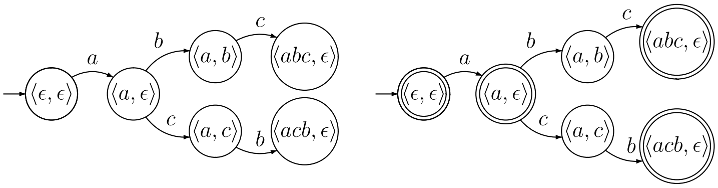
Fig. 4 shows an edit automaton enforcing the property $P = \{a, abc, acb\}$ and the corresponding Büchi automaton.

6.4.1. Local testability decision algorithms

Many algorithms have been proposed to decide whether a given property is LT or not. If the property is LT, then these algorithms can estimate the optimal locality order of the property, i.e., the smallest k for which the

[9] If EM-enforceable security policies are defined only over finite sequences then finite automata can be used instead of Büchi automata.

[10] We use the definition of a Büchi automaton provided in [24] since it allows recognizing finite and infinite sequences.



$\langle\epsilon,\epsilon\rangle$
a
$\langle a,\epsilon\rangle$
b
$\langle a,b\rangle$
c
$\langle abc,\epsilon\rangle$
c
$\langle a,c\rangle$
b
$\langle acb,\epsilon\rangle$

Fig. 4. An edit automaton and the corresponding Büchi automaton.

property can be specified as LT. The algorithms deciding whether a property is LT or not, have a polynomial-time complexity. However, finding the precise order of local testability is proved to be NP-hard. Fortunately, many algorithms have been proposed to provide a locality order estimation for a given LT property. These algorithms are usually, polynomial-time hard.

The implementations of many LT decision algorithms can be found in the TESTA package [32], implemented using C/C^{++}. The existing algorithms can be classified into two main categories, depending on the model used to specify the analyzed property. The first class is based on analyzing properties specified by deterministic finite automata while the second class is based on analyzing properties specified by syntactic semigroups:

(1) The algorithms analyzing properties specified by deterministic finite automata start by reducing the analyzed automata. Reducing an automaton consists in modifying it in such a way that each path not used to recognize the property sequences is suppressed from the automaton transition graph. As examples of the algorithms belonging to this category, we can cite:

- The algorithm provided in [13] to decide whether a given property is LT or not. The time-complexity of this algorithm is $o(|\Sigma|n^2|)$ where Σ is the automaton alphabet and n is the cardinality of the automaton state set.
- The two algorithms provided in [21]. The first algorithm decides whether a given property is strictly LT or not. The time-complexity of this algorithm is $o(|\Sigma|^2mn|)$ where Σ, m, and n are, respectively, the alphabet, the number of edges and the number of states of the finite automaton specifying the property. The second algorithm estimates the locality-order of a given LT property. This algorithm has the same time-complexity as the first algorithm.

(2) The algorithms analyzing properties specified by syntactic semigroups are defined on the basis of semigroups theory. A syntactic semigroup is a semigroup associated with some automaton where the structure of the syntactic semigroup is derived from the transition graph structure of the automaton. Associating a semigroup to some automaton allows verifying some property on the syntactic semigroup instead of verifying it on the original automaton. This approach is motivated by the fact that many results can be verified more easily using semigroups theory than by using automata theory. As examples of the algorithms belonging to this category, we can cite [29] and [27].

6.5. Local testability-based approach for memory-constrained enforcement

The results provided in this Sections 6.1, 6.2, 6.3, and 6.4 allow us to define a general approach for identifying security policies enforceable by memory-constrained EM (Fig. 5). The main procedures used in this approach are the following:

A If the security policy is known to be LT, then by Theorems 6.11, 6.12, and 6.10, we can check if the property is enforceable by memory-constrained CEM (BSA-enforceable), and by Theorems 6.16, 6.17, and 6.15 we can check if the property is enforceable by memory-constrained RWEM (BEA-enforceable).

B If the property is specified by a Büchi automaton, then we first reduce the automaton. After, we call the decision algorithm, if it is based on deterministic automata. Otherwise, we produce the syntactic semigroup and then call the decision algorithm. If the algorithm reveals that the property is LT then procedure A is

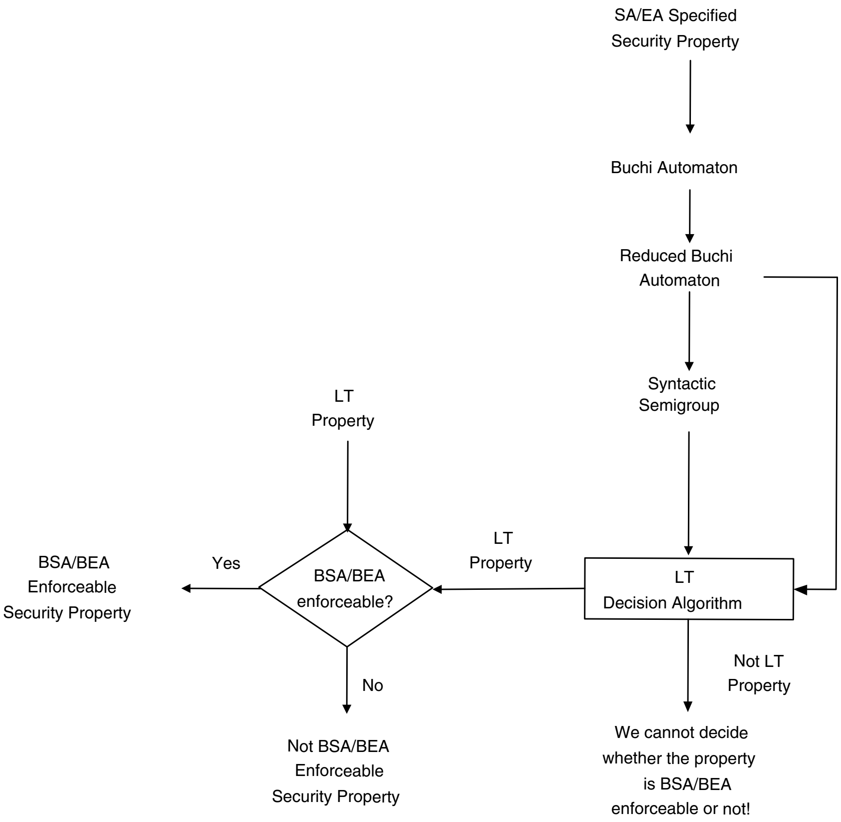

SA/EA Specified Security Property
Buchi Automaton
Reduced Buchi Automaton
Syntactic Semigroup
LT Property
LT Decision Algorithm
LT Property
BSA/BEA enforceable?
Yes
BSA/BEA Enforceable Security Property
No
Not BSA/BEA Enforceable Security Property
Not LT Property
We cannot decide whether the property is BSA/BEA enforceable or not!

Fig. 5. Identifying BHA-enforceable locally testable properties.

used to check if the property is BSA/BEA-enforceable. Otherwise, the algorithm cannot decide whether the property is BSA/BEA-enforceable or not.

C If the property is specified by a SA or an EA, then we can construct the corresponding Büchi automaton by using one of the construction techniques presented in the proofs of Propositions 6.23 or 6.24. Once constructed, the result automaton is minimized and sent to procedure B.

7. Conclusion and future work

In this paper, we propose a characterization of the security policies that are enforceable by execution monitors constrained by memory limitations. The work presented here, is in the same line as the research work advanced by Schneider [25], Ligatti et al. [1,18] and Fong [9] which addresses security policy enforcement. Our approach gives rise to a realistic evaluation of the enforcement power of execution monitoring. This evaluation is based on bounding the memory size used by the monitor to save execution history, and identifying the security policies enforceable under such constraint.

Our contribution in characterizing memory-constrained EM is mainly threefold. First, we instantiated and extended an abstraction based on memory limitation to security automata [25] 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*. Second, we identified 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 reasoned about the memory-constrained enforcement power by investigating the enforcement of locally testable properties [3], a well studied class of languages recognizable by investigating local information. Among the existing locally testable properties classes, we identified (1) those classes that can be enforced by BHA, (2) those that are not BHA-enforceable at all, and (3) those that can be BHA-enforceable under some conditions and we precisely identified such conditions. These results were reinforced by pointing out algorithms that can decide whether a security policy is locally testable or not. This allowed us to define a general approach for deciding whether a given security policy is enforceable by memory-constrained EM. This approach is based on existing algorithms which decide locally testable properties.

As future work, we plan to:

- Design and implement tools for specifying security policies enforceable by memory-constrained EM. These tools will be based on selecting the best algorithms identifying locally testable properties and adapting them to EM-enforceable properties. This will improve our BHA-enforceable security policies classification by identifying new classes of practical EM-enforceable policies.
- Define new characterizations of constrained-EM based on language theory results. These characterizations will cover different kinds of constraints and, hence, different kinds of execution abstractions. In additions, these characterizations should associate EM-enforceable policy classes to language theory classes.
- Generalize our investigation to other classes of security enforcement mechanisms, namely, static analysis and program rewriting.

Appendex A. Proofs of Theorems

Proposition 6.2. *Let k be any positive integer, and let $F \subseteq \Sigma_k$, $P, S \subseteq \Sigma_{k-1}$, and $X \subseteq \Sigma_{\leqslant k-1}$ be the sets used to define a strictly 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 following 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 $L' = X \cup (F_{initial} \cap F_{terminal}) = L \cap \Sigma_{\leqslant k}$ and $L'' = \{\sigma \in L : \sigma isinfiniteor \quad |\sigma| > k\}$ 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, i.e., $F_R \subseteq F_{terminal}$, then 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 us 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 us assume 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 cannot end with a factor 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). □

Proposition 6.4. *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 $X \subseteq \Sigma_{\leqslant k-1}$. 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) $|\sigma| \leqslant k$. Obviously, $\sigma \in X \cup P$. Since $X \cup P$ is prefix-closed then we have $Pref(\sigma) \subseteq (X \cup P) \subset L$.
  (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 $Pref(P)$.

- *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 $\exists \sigma \in X \cup P. \exists \sigma' \in Pref(\sigma). \sigma' \notin X \cup P$. 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). □

Proposition 6.7. Let k be any positive integer, and let P, S, X be three sets such that $P, S \subseteq \Sigma_k$, and $X \subseteq \Sigma_{\leqslant k-1}$. If L is a k-ST property defined by the two sets S and X or a k-PST property defined by the sets P, S, and X, then L is not prefix-closed.

Proof. Since we can extend any 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 the sequence σs where s is any element of S. Similarly, for any element p of P and any sequence σ such that $p\sigma \notin L'$, $p\sigma$ can be extended to the sequence $p\sigma s \in L'$ where s is any element of S. Therefore, L' is not prefix-closed. □

Proposition 6.9. Let k be any positive integer, and let $F \subseteq \Sigma_k$. If L is a k-SLT property defined by the set F, then L is prefix-closed.

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

Theorem 6.11. *Let k be any positive integer. Any prefix-closed k-PT property L that is defined according to Definition 6.3 is enforceable by some k-BSA.*

Proof. From Proposition 6.4, L is prefix-closed if and only if $X \cup P$ is prefix-closed. The k-BSA enforcing L is defined by $A = \langle \Sigma, Q = \Sigma_{\leqslant k}, \epsilon, \delta \rangle$ 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* $|\sigma| \leqslant k$, *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: $\epsilon \xrightarrow{\sigma[1]} \sigma[1] \ldots \sigma[m-1] \xrightarrow{\sigma[m]} \sigma$ where $|\sigma| = m$. It is easy to see that no sequence of $\Sigma_{\leqslant k} \setminus (X \cup P)$ 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 $\epsilon \xrightarrow{\sigma'[1]} \sigma'[1] \ldots \sigma'[m\ -1\ ] \xrightarrow{\sigma'[m]} \sigma' \xrightarrow{\sigma''[1]} \sigma' \ldots \sigma' \xrightarrow{\sigma''[d]} \sigma'$ 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 6.12. Let k be any positive integer. Any k-SLT property L that is defined according to Definition 6.8 is enforceable by some k-BSA.

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

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

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 $\Sigma_{\leqslant k-1}$ is recognizable by the path $\epsilon \xrightarrow{\sigma[1]} \sigma[1] \ldots \sigma[m\ -1\ \ ] \xrightarrow{\sigma[m]} \sigma$ where $|\sigma| = m$.

(2) *Case* σ infinite or $|\sigma| \geqslant k$: 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 $\epsilon \xrightarrow{\sigma'[1]} \sigma'[1] \ldots \xrightarrow{\sigma'[k-1]} \sigma' \xrightarrow{\sigma''[1]} f_1 \xrightarrow{\sigma''[2]} f_2 \ldots f_{d-1} \xrightarrow{\sigma''[d]} f_d$ where $|\sigma''| = d$ and $f_1 = \sigma'[2..]\sigma''[1]$ and $\forall\, 1 < i \leqslant d.\ f_i = f_{i-1}[2..]\sigma''[i]$ and $\forall\, 1 \leqslant i \leqslant d. f_i \in F$. Thus all the factors of length k of σ are in F. □

Theorem 6.16. Let k be any positive integer. Any k-PT property is enforceable by some k-BEA.

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

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

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 $|\sigma| \leqslant k$, 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 $\langle\epsilon,\epsilon\rangle \underset{[\tau_1]}{\overset{\sigma[1]}{\rightarrow}} \langle\sigma_{1Acc},\sigma_{1Sup}\rangle \underset{[\tau_2]}{\overset{\sigma[2]}{\rightarrow}} \ldots \underset{[\tau_{m-1}]}{\overset{\sigma[m-1]}{\rightarrow}} \langle\sigma_{m-1Acc},\sigma_{m-1Sup}\rangle \underset{[\tau_m]}{\overset{\sigma[m]}{\rightarrow}} \langle\sigma,\epsilon\rangle$ where:
$|\sigma| = m$ and $\forall i.1 < i < m.\tau_i = \delta(\langle\sigma_{i-1Acc},\sigma_{i-1Sup}\rangle, \sigma[i])$, and $\forall i.1 \leqslant i < m$, the state $\langle\sigma_{iAcc},\sigma_{iSup}\rangle$ satisfies:

  (a) $\sigma_{iAcc}\sigma_{iSup} = \sigma[..i] \wedge (\sigma[..i] \in P \cup X \Rightarrow \sigma_{iSup} = \epsilon)$,
  (b) $(\langle\sigma_{i+1Acc},\sigma_{i+1Sup}\rangle, \tau_{i+1}) = \delta(\langle\sigma_{iAcc},\sigma_{iSup}\rangle, \sigma[i])$.
  (c) $(\langle\sigma_{1Acc},\sigma_{1Sup}\rangle, \tau_1) = \delta(\langle\epsilon,\epsilon\rangle, \sigma[1])$.

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 $\langle\epsilon,\epsilon\rangle \underset{[\tau_1]}{\overset{\sigma'[1]}{\rightarrow}} \ldots \rightarrow^{\sigma'[m]}_{[\tau_m]} \langle\sigma',\epsilon\rangle \underset{[\sigma''[1]]}{\overset{\sigma''[1]}{\rightarrow}} \langle\sigma',\epsilon\rangle \ldots \langle\sigma',\epsilon\rangle \underset{[\sigma''[d]]}{\overset{\sigma''[d]}{\rightarrow}} \langle\sigma',\epsilon\rangle$ 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 cannot be recognized by the automaton A. □

Theorem 6.17. Let k be any positive integer. Any k-SLT property 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-SLT property L over Σ^∞. The k-BEA enforcing L is defined by $A = \langle\Sigma, Q = (\Sigma_{\leqslant k} \times \Sigma_{\leqslant k})_{\leqslant k}, \langle\epsilon, \epsilon\rangle, \delta\rangle$ where the transition function $\delta : \{\langle\sigma, \epsilon\rangle | \sigma \in \Sigma_{\leqslant k}\} \times \Sigma \rightarrow \{\langle\sigma, \epsilon\rangle | \sigma \in \Sigma_{\leqslant k}\} \times \Sigma^*$ is defined by the following:

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

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 rule (1) ensures that any such state is reachable by the path $\langle\epsilon, \epsilon\rangle \underset{[\sigma[1]]}{\overset{\sigma[1]}{\rightarrow}} \langle\sigma[..1], \epsilon\rangle \underset{[\sigma[2]]}{\overset{\sigma[2]}{\rightarrow}} \ldots \underset{[\sigma[m-1]]}{\overset{\sigma[m-1]}{\rightarrow}} \langle\sigma[..m-1], \epsilon\rangle \underset{[\sigma[m]]}{\overset{\sigma[m]}{\rightarrow}} \langle\sigma, \epsilon\rangle$.

(2) Case $|\sigma| \geqslant k$: By rule (1) and rule (2), any sequence having all its factors of length k in F is recognizable. By Definition 6.8, 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: $\langle\epsilon, \epsilon\rangle \underset{[f[1]]}{\overset{f[1]}{\rightarrow}} \langle f[..1], \epsilon\rangle \underset{[f[2]]}{\overset{f[2]}{\rightarrow}} \ldots \underset{[f[k-1]]}{\overset{f[k-1]}{\rightarrow}} \langle f[..k-1], \epsilon\rangle \underset{[f[k]]}{\overset{f[k]}{\rightarrow}} \langle f, \epsilon\rangle \underset{[\sigma''[1]]}{\overset{\sigma''[1]}{\rightarrow}} \langle f_1, \epsilon\rangle \underset{[\sigma''[2]]}{\overset{\sigma''[2]}{\rightarrow}} \ldots \underset{[\sigma''[m-1]]}{\overset{\sigma''[m-1]}{\rightarrow}} \langle f_{m-1}, \epsilon\rangle \underset{[\sigma''[m]]}{\overset{\sigma''[m]}{\rightarrow}} \langle f_m, \epsilon\rangle$ where for all integer i such that $1 \leqslant i \leqslant m.\ f_i \in F \wedge f_i \in Suf(f\sigma''[..i])$.

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

Theorem 6.19. If $|\Sigma| > 1$ then any PST property defined over Σ^∞ is not BEA-enforceable.

Proof. We have to prove that if $|\Sigma| > 1$ then for any PST property, there is no BEA enforcing it. We proceed by contradiction. For some positive integer k, let L be any k-PST property defined by the three sets $P \subseteq \Sigma_k$, $S \subseteq \Sigma_k$ and $X \subseteq \Sigma_{\leqslant k-1}$. Let us suppose that there exists a BEA $A = \langle\Sigma, Q \subseteq (\Sigma_{\leqslant k'} \times \Sigma_{\leqslant k'})_{\leqslant k'}, q_0, \delta\rangle$ of bound k' enforcing L. However, we can find a sequence $\sigma \in L$ that is not recognizable by A. Such sequence can be any sequence $\sigma = \sigma'\sigma''s$ where (1) s is any suffix from S, (2) $\sigma' \in L$, (3) $Fact_k(\sigma'') \cap S = \emptyset$, and (4) $|\sigma''| > k'$. Intuitively, in order to recognize σ, we need to suppress the entire subsequence σ'' and save 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 BEA enforcing L. The existence of the sequence σ'' is guaranteed by the fact that (a) $|\Sigma| > 1$ and (b) L is really specifying a PST property. Indeed, if $\neg(|\Sigma| > 1)$, i.e., $|\Sigma| = 1$, e.g., $\Sigma = \{a\}$ then $P = a^k\{a\}^\infty \cap (\{a\}^*a^k)^\infty \cup X = a^k a^\infty \cup X$ which is a prefix testable property! In this case, the property L is studied as PST while, in reality, it specifies a prefix-testable property. For the case $|\Sigma| > 1$ and L is really specifying a PST property, the existence of σ'' is ensured. Indeed, if there is no sequence σ'' satisfying (3) and (4) then there exists some integer $k'' \leqslant k'$ and some set $R \subseteq \Sigma_{\leqslant k''}$ such that $L = (RS)^\infty \cup X$. Let F be the set defined by $Fact_k(\{\sigma s |\ \sigma \in R \wedge s \in S\}) \cup Fact_k(\{s\sigma | \sigma \in R \wedge s \in S\})$. Then, the property L can be viewed as a strictly k-locally testable property defined by the sets P, S, X, and F. Indeed $L = ((P\Sigma^\infty \cap (\Sigma^*S)^\infty) \setminus \Sigma^*\overline{F}\Sigma^\infty) \cup X$ where $\overline{F} = \Sigma^k \setminus F$. Thus the property L is studied as PST property while in reality, it is specifying a strictly locally testable property which is in contradiction with (b). □

Theorem 6.20. Let k be any positive integer. Any k-ST property L 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:

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

Theorem 6.21. Let k be any positive integer. Any k-PST property L 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:

$$\forall \langle \sigma_{Acc}, \sigma_{Sup} \rangle \in \Sigma^* \times \Sigma^*. \forall a \in \Sigma.$$
$$\delta(\langle \sigma_{Acc}, \sigma_{Sup} \rangle, a) = \begin{cases} (\langle \sigma_{Acc}, \sigma_{Sup}a \rangle, \epsilon) \text{ if } \sigma_{Acc}\sigma_{Sup}a \notin X \wedge \\ (Pref(\sigma_{Acc}\sigma_{Sup}a) \cap P = \emptyset \vee Suf(\sigma_{Acc}\sigma_{Sup}a) \cap S = \emptyset) \ (1) \\ (\langle \sigma_{Acc}\sigma_{Sup}a, \epsilon \rangle, \sigma_{Sup}a) \text{ if } \sigma_{Acc}\sigma_{Sup}a \in X \vee \\ (Pref(\sigma_{Acc}\sigma_{Sup}a) \cap P \neq \emptyset \wedge Suf(\sigma_{Acc}\sigma_{Sup}a) \cap S \neq \emptyset) \ (2) \end{cases} \quad \square$$

References

[1] L. Bauer, J. Ligatti, D. Walker, More enforceable security policies, in: Foundations of Computer Security, Copenhagen, Denmark, 2002, pp. 95–104.

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

[3] D. Beauquier, J.E. Pin, Languages and scanners, Theor. Comput. Sci. 84 (1991) 3–21.

[4] K. Biba, Integrity considerations for secure computer systems, Technical Report 76-372, US Air Force Electronic Systems Division, 1977.

[5] W.E. Boebert, R.Y. Kain, A practical alternative to hierarchical integrity policies, in: Proceedings of the 8th National Computer Security Conference, 1985, pp. 18–27.

[6] D.F.C. Brewer, M.J. Nash, The Chinese wall security policy, in: IEEE Symposium on Security and Privacy, 1989, pp. 206–214.

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

[8] G. Edjladi, A. Acharya, V. Chaudhary, History-based access control for mobile code, in: 5th ACM Conference on Computer and Communications Security, San Francisco, CA, USA, 1998, pp. 38–48.

[9] P.W.L. Fong, Access control by tracking shallow execution history, in: Proceedings of the 2004 IEEE Symposium on Security and Privacy, Berkeley, CA, 2004.

[10] C. Fournet, A.D. Gordon, Stack inspection: theory and variants, in: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press, 2002, pp. 307–318.

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

[12] K.W. Hamlen, G. Morrisett, F.B. Schneider, Computability classes for enforcement mechanisms, ACM Trans. Program. Lang. Syst. 28 (1) (2006) 175–205.

[13] S. Kim, R. McNaughton, R. McCloskey, A polynomial time algorithm for the local testability problem of deterministic finite automata, IEEE Trans. Comput. 40 (1991) 1087–1093.

[14] S. Kim, R. McNaughton, Computing the order of a locally testable automaton, SIAM J. Comput. 23 (1994) 1193–1215.

[15] O. Kupferman, Y. Lustig, M.Y. Vardi, On locally checkable properties, in: Proceedings of the 13th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR 2006), 2006.

[16] L. Lamport, Proving the correctness of multiprocess programs, IEEE Trans. Software Eng. 3 (2) (1977) 125–143.

[17] B. Lampson, Protection, in: Proceedings 5th Symposium on Information Sciences and Systems, Princeton, New Jersey, 1971, pp. 437–443.

[18] J. Ligatti, L. Bauer, D. Walker, Edit automata: enforcement mechanisms for run-time security policies, Int. J. Inform. Secur. 4 (1–2) (2005) 2–16 (Published online 26 Oct 2004).

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

[20] J. Ligatti, L. Bauer, D. Walker, Enforcing non-safety security policies with program monitors, in: Computer Security—ESORICS 2005: 10th European Symposium on Research in Computer Security, in: Lecture Notes in Computer Science, vol. 3679, 2005, pp. 355–373.

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

[22] W.H Paxton, Proceedings of the Seventh ACM Symposium on Operating Systems Principles, ACM Press, 1979, pp. 18–23.

[23] D. Perrin, J.-E. Pin, Infinite words. Automata, semigroups, logic and games, Pure and Applied Mathematics, vol. 141, Elsevier, 2004.

[24] J.E Pin, Finite semigroups and recognizable languages : an introduction, in: J. Fountain (Ed.), NATO Advanced Study Institute Semigroups, Formal Languages and Groups, Kluwer academic publishers, 1995, pp. 1–32.

[25] F.B. Schneider, Enforceable security policies, ACM Trans. Inform. Syst. Secur. 3 (1) (2000) 30–50.

[26] A. Taivalsaari, Java Specification Request 139 J2ME Connected Limited Device Configuration 1.1, March 2003.

[27] A.N. Trahtman, A polynomial time algorithm for local testability and its level, J. Algebra Comput. 9 (1) (1998) 1–2.

[28] A.N. Trahtman, An algorithm to verify local threshold testability of deterministic finite automata, Lecture Notes Comput. Sci. 2214 (2001) 164–171.

[29] A.N. Trahtman, A polynomial time algorithm for left [right] local testability, Lecture Notes Comput. Sci. 2608 (2003) 203–212.

[30] M. Viswanathan, Foundations for the Run-time Analysis of Software Systems, Ph.D. thesis, University of Pennsylvania, 2000.

[31] W.D. Young, P.A. Telega, W.E. Boebert, A verified labler for the secure ada target, in: In Proceedings of the 9th National Computer Security Conference, 1986, pp. 55–61.

[32] A Package TESTAS for Checking Some Kinds of Testability.