available at www.sciencedirect.com

ScienceDirect

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

Computers & Security

Extending the enforcement power of truncation monitors using static analysis

Hugues Chabot, Raphaël Khoury, Nadia Tawbi*

Département d'informatique et de génie logiciel, Université Laval, 1065, av. de la Médecine, Québec City, Québec, Canada G1V 0A6

ARTICLE INFO

Article history:
Received 29 June 2010
Received in revised form
23 October 2010
Accepted 21 November 2010

Keywords:
Computer security
Dynamic analysis
Monitoring
Software safety

ABSTRACT

Runtime monitors are a widely used approach to enforcing security policies. Truncation monitors are based on the idea of truncating an execution before a violation occurs. Thus, the range of security policies they can enforce is limited to safety properties. The use of an a priori static analysis of the target program is a possible way of extending the range of monitorable properties. This paper presents an approach to producing an in-lined truncation monitor, which draws upon the above intuition. Based on an a priori knowledge of the program behavior, this approach allows, in some cases, to enforce more than safety properties and is more powerful than a classical truncation mechanism. We provide and prove a theorem stating that a truncation enforcement mechanism considering only the set of possible executions of a specific program is strictly more powerful than a mechanism considering all the executions over an alphabet of actions.

1. Introduction

Execution monitoring is an approach to enforcing security policies that seeks to allow an untrusted code to run safely by observing its execution and reacting if necessary to prevent a potential violation of a user-supplied security policy. This method has many promising applications, particularly with respect to the safe use of mobile code.

Academic research on monitoring has generally focused on two questions. The first relates to the set of policies that can be enforced by monitors and the conditions under which this set could be extended. The second question deals with the way to in-line a monitor in an untrusted or potentially malicious program in order to produce a new instrumented program that provably respects the desired security policy.

While studies on security policy enforcement mechanisms show that an a priori knowledge of the target program's behavior would increase the power of these mechanisms (Hamlen et al., 2006; Bauer et al., 2002), no further investigations have been pursued in order to take full advantage of this idea in the context of runtime monitoring. As a result, implementations of truncation based monitoring frameworks remain limited in the set of policies that they can enforce to the set of safety properties.

This paper, presents an approach to generate a safe instrumented program, from a security policy and an untrusted program in which the monitor draws on an a priori knowledge of the program's possible behavior. This allows the monitor to sometimes enforce non-safety properties, which were beyond the scope of previous approaches.

This approach draws on advances in discrete events system control by Ramadge and Wonham (1989) and on related subsequent research by Langar and Mejri (2005) and consists in combining two models via the automata product operator: a model representing the system's behavior and another one representing the property to be enforced. In this

* *Corresponding author*. Tel.: +1 418 653 0177.
E-mail addresses: hugues.chabot.1@ulaval.ca (H. Chabot), raphael.khoury.1@ulaval.ca (R. Khoury), nadia.tawbi@ift.ulaval.ca (N. Tawbi).
0167-4048/$ – see front matter © 2010 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cose.2010.11.004

approach, the system's behavior is modeled by an LTS and the property to be enforced is stated as a Rabin automaton, a model which can recognize the same class of languages as non-deterministic Büchi automata (Perrin and Pin, 2004). A major advantage of this representation over alternatives such as the Büchi automaton is its determinism, which simplifies the method and the associated proofs.

The algorithm either returns an instrumented program that provably respects the input security policy, otherwise it terminates with an error message. While the latter case sometimes happens, it is important to stress that this will never occur if the desired property is a safety property which can be enforced using existing approaches. Indeed, any safety property is still enforced by truncation as is presently the case in preceding works. When enforcing a non-safety property, the approach relies both on static program transformations and runtime truncation. The approach presented in this paper is thus strictly more expressive. Furthermore, this increase in the set of enforceable property is achieved without relying on the transformative capacities of the edit automaton, and without imposing a runtime overhead to the execution.

The rest of this paper is organized as follows. Section 2 presents a review of related work. Section 3 defines some concepts that are used throughout the paper. The elaborated method is presented in Section 4. Section 5 discusses the theoretical underpinnings of the method. Some concluding remarks are finally drawn in Section 6 together with an outline of possible future work.

2. Related work

Schneider, in his seminal work (Schneider, 2000), was the first to investigate the question of which security policies could be enforced by monitors. He focused on specific classes of monitors, which observe the execution of a target program with no knowledge of its possible future behavior and with no ability to affect it, except by aborting the execution. Under these conditions, he found that a monitor could enforce precisely those security policies that are identified in the literature as safety properties, and are informally characterized by prohibiting a certain bad thing from occurring in a given execution. These properties can be modeled by a security automaton and their representation has formed the basis of several practical as well as theoretical monitoring frameworks.

Schneider's study also suggested that the set of properties enforceable by monitors could be extended under certain conditions. Building on this insight, Bauer et al. (2002) and Ligatti et al. (2005a) examined the way the set of policies enforceable by monitors would be extended if the monitor had some knowledge of its target's possible behavior or if its ability to alter that behavior were increased. The authors modified the above definition of a monitor along three axes, namely (1) the means on which the monitor relies in order to respond to a possible violation of the security policy; (2) whether the monitor has access to information about the program's possible behavior; (3) and how strictly the monitor is required to enforce the security policy. Consequently, they were able to provide a rich taxonomy of classes of security policies, associated with the appropriate model needed to enforce them. Several of these models are strictly more powerful than the security automata developed by Schneider and are used in practice.

Evolving along this line of inquiry, Ligatti et al. (2005b) gave a more precise definition of the set of properties enforceable by the most powerful monitors, while Fong (2004) and Talhi et al. (2008) expounded on the capabilities of monitors operating under memory constraints. Hamlen et al. (2006), on the other hand showed that in-lined monitors (whose operation is injected into the target program's code, rather than working in parallel) can also enforce more properties than those modeled by a security automaton. In Bauer et al. (2006), a method is given to enforce both safety and co-safety properties by monitoring. Alternative definitions of enforcement were given in Bielova et al. (2009), Khoury and Tawbi (in press) and Ligatti and Reddy (2010).

The first practical application using this framework was developed by Erlingsson and Schneider (2000). In that project, a security automaton is merged into object code, and static analysis is used to reduce the runtime overhead incurred by the policy enforcement. Similar approaches, working on source code, were developed by Colcombet and Fradet (2000), by Langar and Mejri (2005) and by Kim (2001), Kim et al. (2004), Lee et al. (1999), Sokolsky et al. (1999). All these methods are limited to enforcing safety properties, which must be included either as a security automaton, or stated in a custom logic developed for this application. The first two focus on optimizing the instrumentation introduced in the code.

3. Preliminaries

Before moving on, let us briefly start with some preliminary definitions.

The desired security property is stated as a Rabin automaton. The Rabin automaton is a finite state automaton which can recognize infinite-length sequences. The input sequence is recognized if it satisfies an acceptance condition, given by a set of pairs of sets of states, which restricts the states which an accepted sequence can visit infinitely often. When the Rabin automaton models a security property, the accepted sequences correspond to the sequences satisfying the property. A Rabin automaton $\mathcal{R}$, over alphabet $\mathcal{A}$ is a tuple (Q,q_0,δ,C) such that

- $\mathcal{A}$ is a finite or countably infinite set of symbols;
- Q is a finite set of states;
- $q_0 \in Q$ is the initial state;
- $\delta \subseteq Q \times A \times Q$ is a transition function;
- $C = \{(L_j,U_j)|j \in J\}$ is the acceptance set. It is a set of couples (L_j,U_j) where $L_j \subseteq Q$ and $U_j \subseteq Q$ for all $j \in J$ and $J \subseteq \mathbb{N}$.

Let $\mathcal{R}$ stand for a Rabin automaton defined over alphabet $\mathcal{A}$. A subset $Q' \subseteq Q$ is *admissible* if and only if there exists a $j \in J$ such that $Q' \cap L_j = \emptyset$; and $Q' \cap U_j \neq \emptyset$.

For the sake of simplicity, the elements defining an automaton or a model are referred to using the following formalism: the set of states Q of automaton $\mathcal{R}$ is referred to as $\mathcal{R}\cdot Q$ or simply Q when $\mathcal{R}$ is clear from the context.

A *path* π, is a finite (respectively infinite) sequence of states $\langle q_1,q_2,\ldots,q_n\rangle$ (respectively $\langle q_1,q_2,\ldots\rangle$) such that there exists a finite (respectively infinite) sequence of symbols $a_1,a_2,\ldots,a_n$

(respectively $a_1,a_2,\ldots$) called the label of π such that $\delta(q_i,a_i) = q_{i+1}$ for all $i \in \{0,\ldots,n\}$ (respectively $i \geq 0$). In fact, a path is a sequence of states consisting of a possible run of the automaton, and the label of this path is the input sequence that generates this run. A path is said to be empty if its label is the empty sequence ϵ.

Let set(π) denote the set of states visited by the path π. The first state of π is called the origin of π. If π is finite, the last state it visits is called its end; otherwise, if it is infinite, inf(π) stands for the set of states that are visited infinitely often in π. A path π is *initial* if and only if its origin is q_0, the initial state of the automaton, and it is *final* if and only if it is infinite and inf(π) is admissible. In framework under consideration, a program execution may be of infinite length representing the executions of programs such as daemons or servers. Since the Rabin automaton is an automaton over infinite sequences, while the target program might contain both finite and infinite sequences, finite sequences are transformed into equivalent infinite ones by appending to each valid sequence an infinite repetition of a void action, not present in the input alphabet.

A path is *successful* if and only if it is both initial and final. The property of successfulness of a path determines, in fact, the acceptance condition of Rabin automata. A sequence is accepted by a Rabin automaton *iff* it is the label of a *successful* path.

The set of all accepted sequences of $\mathcal{R}$ is the language recognized by $\mathcal{R}$, noted $\mathcal{L}_\mathcal{R}$.

Let $q \in Q$ be a state of $\mathcal{R}$. A state q is said to be accessible *iff* there exists an initial path (possibly the empty path) that visits q; q is co-accessible *iff* it is the origin of a final path.

Executions are modeled as sequences of atomic actions taken from a finite or countably infinite set of actions $\mathcal{A}$. The empty sequence is noted ϵ, the set of all finite length sequences is noted $\mathcal{A}^*$, that of all infinite-length sequences is noted $\mathcal{A}^\omega$, and the set of all possible sequences is noted $\mathcal{A}^\infty = \mathcal{A}^\omega \cup \mathcal{A}^*$. Let $\tau \in \mathcal{A}^*$ and $\sigma \in \mathcal{A}^\infty$ be two sequences of actions. $\tau;\sigma$ denotes the concatenation of τ and σ. τ is said to be a prefix of σ noted $\tau \preceq \sigma$ *iff* $\tau \in \mathcal{A}^*$ and there exists a sequence σ' such that $\tau;\sigma' = \sigma$.

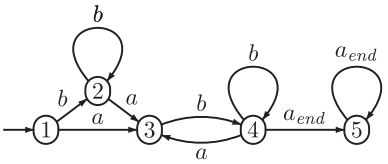
Let $a \in \mathcal{A}$ be an action symbol. A state $q' \in Q$ is an a-successor of q if $\delta(q,a) = q'$. Conversely, a state q' is a successor of q if there exists a symbol a such that $\delta(q,a) = q'$.

Let $\pi = q_1,q_2,\ldots,q_n$ be a finite path in $\mathcal{R}$. This path is a cycle if $q_1 = q_n$.

The cycle π is admissible *iff* set(π) is admissible. It is accessible *iff* there is a state q in set(π) such that q is accessible, and likewise, it is co-accessible *iff* there is a state q in set(π) such that q is co-accessible.

An example of a Rabin automaton is given in Fig. 1. In this automaton, all states are both accessible and co-accessible. The paths $\langle 3,4,3,4,3\rangle$, $\langle 3,4,3\rangle$ and $\langle 2,2\rangle$ are inadmissible cycles, while $\langle 5,5\rangle$ is an admissible cycle and both infinite paths $\langle 1,2,3,4,5,5,\ldots\rangle$ and $\langle 1,2,3,4,3,4,4,\ldots\rangle$ are initial and final and therefore both are successful.

Finally a security property $\widehat{\mathcal{P}}$ is a predicate on executions. An execution σ is said to be valid or to respect the property if $\widehat{\mathcal{P}}(\sigma)$. A Rabin automaton $\mathcal{R}$ represents a security policy $\widehat{\mathcal{P}}$ *iff* $\mathcal{L}_\mathcal{R} = \{\sigma | \widehat{\mathcal{P}}(\sigma)\}$, the set of executions that satisfy the security policy. Abusing the notation, the application of $\widehat{\mathcal{P}}$ is extended to a set of sequences, thus if Σ is a set of sequences $\widehat{\mathcal{P}}(\Sigma)$ means that all the sequences of Σ satisfy $\widehat{\mathcal{P}}$.



$$C = \{(\{3\},\{4\}),(\emptyset,\{5\})\}$$

Fig. 1 – A Rabin automaton with acceptance condition C.

4. Method

This section explains the approach in more detail and illustrates its operation with an example. The main algorithm takes as input a Rabin automaton $\mathcal{R}$, which represents a security Policy $\widehat{\mathcal{P}}$ and a labeled transition system (LTS) $\mathcal{M}$, which models a program. The algorithm either returns a model of an instrumented program that enforces $\widehat{\mathcal{P}}$ on $\mathcal{M}$ or returns an error message. The latter case occurs when it is not possible to produce an instrumented program that both enforce the desired security property and generate all valid sequences of $\mathcal{M}$.

In concordance with Erlingsson (2004), Hamlen et al. (2006) and Ligatti et al. (2005a), an enforcement mechanism successfully enforces the property if the two following conditions are satisfied. First, the enforcement mechanism must be transparent; meaning that all possible program executions that respect the property must be emitted, i.e. the enforcement mechanism cannot prevent the execution of a sequence satisfying the property. Second, the enforcement mechanism must be sound, meaning that it must ensure that all observable output respects the property. These ideas are revisited and expanded in Sections 4.3.2 and 5. Each step of the approach is illustrated using an example program and a security policy.

4.1. *Property encoding*

As mentioned earlier, the desired security property is stated as a Rabin automaton. The security property $\widehat{\mathcal{P}}$ to which the target program must conform is modeled by the Rabin automaton in Fig. 1, over the alphabet $\mathcal{A} \cup \{a_{\text{end}}\}$ with $\mathcal{A} = \{a,b\}$. The symbol a_{end} is a special token added to $\mathcal{A}$ to capture the end of a finite sequence, since the Rabin automaton only accepts infinite-length sequences. The finite sequence σ is thus modeled as $\sigma;(a_{\text{end}})^\omega$. The language accepted by this automaton is the set of executions that contains only a finite non-empty number of a actions and such that finite executions end with a b action.

For the sake of simplicity, if a sequence $\sigma = \tau;(a_{\text{end}})^\omega$ with $\tau \in \mathcal{A}^*$ is such that $\widehat{\mathcal{P}}(\sigma)$, we write $\widehat{\mathcal{P}}(\tau)$.

4.2. *Program abstraction*

The program is abstracted as a labeled transition system (LTS). This is a conservative abstraction, widely used in model checking and static analysis, in which a program is abstracted as a graph, whose nodes represent program points, and whose edges are labeled with instructions (or abstractions of instructions, or actions). The LTS representing the program could be built directly from the control flow graph after

a control flow analysis (Aho et al., 1986; Beyer et al., 2007). Formally, an LTS $\mathcal{M}$, over alphabet $\mathcal{A}$ is a deterministic graph (Q,q_0,δ) such that:

- $\mathcal{A}$ is a finite or countably infinite set of actions;
- Q is a finite set of states;
- q_0 is the initial state;
- $\delta : Q \times \mathcal{A} \times Q$ is a transition function. For each $q \in Q$, there must be at least one $a \in \mathcal{A}$ for which $\delta(q,a)$ is defined.

Here also a finite sequence σ is extended with the suffix $(a_{end})^{\omega}$ yielding the infinite sequence σ; $(a_{end})^{\omega}$. Thus only infinite-length executions are considered.

In general, static analysis tools do not always generate deterministic LTSs. Yet, this restriction can be imposed with no loss of generality. Indeed, a non-deterministic LTS $\mathcal{M}$ over alphabet $\mathcal{A}$ can be represented by an equivalent deterministic LTS $\mathcal{M}'$ over alphabet $\mathcal{A} \times \mathbb{N}$, which is equivalent to $\mathcal{M}$ if the numbers $i \in \mathbb{N}$ associated with the actions is ignored. Each occurrence of an action a is associated with a unique index in $\mathbb{N}$ so as to distinguish it from other occurrences of the same action a. In what follows, only deterministic LTSs are thus considered.

The example program used to illustrate the approach is modeled by the LTS in Fig. 2, over the alphabet $\mathcal{A}$. The issue consisting of how to abstract a program into an LTS is beyond the scope of this paper.

As with the Rabin Automata, a *path* π is a finite or infinite sequence of states $\langle q_1,q_2,\ldots\rangle$ such that there exists a corresponding sequence of actions $(a_1,a_2\ldots)$ called the label of π, for which the $\delta(q_i,a_i) = q_{i+1}$.

The set of all labels of infinite paths starting in q_0 is the language generated or emitted by $\mathcal{M}$ and is noted $\mathcal{L}_{\mathcal{M}}$.

4.3. Algorithm

The algorithm's input consists of the program model $\mathcal{M}$ and a Rabin automaton $\mathcal{R}$ which encodes the property. The output is a truncation automaton $\mathcal{T}$ representing a model of an inlined monitored program acting exactly identically to the input program for all the executions satisfying the property and halting a bad execution after producing a valid prefix of this execution.

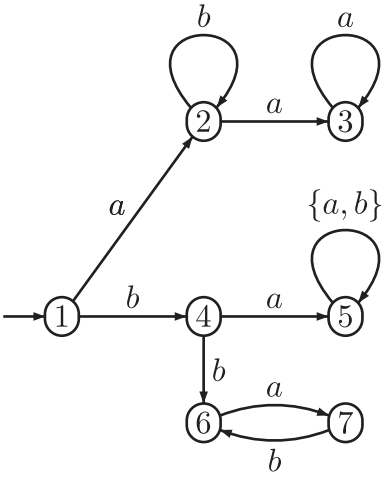


Fig. 2 – Example – Labeled transition system.

A high level description of the algorithm is as follows:

1. Build a product automaton $\mathcal{R}^P$ whose recognized language is exactly: $\mathcal{L}_{\mathcal{R}^P} = \mathcal{L}_{\mathcal{R}} \cap \mathcal{L}_{\mathcal{M}}$.
2. Build $\mathcal{R}^T$ from $\mathcal{R}^P$ by the application of a transformation allowing it to accept partial executions of the program modeled by $\mathcal{M}$ that satisfy the property $\widehat{\mathcal{P}}$.
3. Check if $\mathcal{R}^T$ could be used as a truncation automaton and produce an LTS $\mathcal{T}$ modeling the program instrumented by a truncation mechanism otherwise produce **error**.

The following sections give more details on each step.

4.3.1. *Automata product*

The first phase of the transformation is to construct $\mathcal{R}^P$, a Rabin automaton that accepts the intersection of the language accepted by the automaton $\mathcal{R}$ and the language emitted by $\mathcal{M}$. This is exactly the product of these two automata. Thus $\mathcal{R}^P$ accepts the set of executions that both respect the property and represent executions of the target program.

Given a property automaton $\mathcal{R} = (\mathcal{R}\cdot Q, \mathcal{R}\cdot q_0, \mathcal{R}\cdot\delta, \mathcal{R}\cdot C)$ and a Labeled Transition system $\mathcal{M} = (\mathcal{M}\cdot Q, \mathcal{M}\cdot q_0, \mathcal{M}\cdot\delta)$ the automaton $\mathcal{R}^P$ is constructed as follows:

- $\mathcal{R}^P\cdot Q = \mathcal{R}\cdot Q \times \mathcal{M}\cdot Q$
- $\mathcal{R}^P\cdot q_0 = (\mathcal{R}\cdot q_0, \mathcal{M}\cdot q_0)$
- $\forall_{q \in R\cdot Q, q' \in \mathcal{M}\cdot Q \wedge a \in (A \cup \{a_{end}\})}$

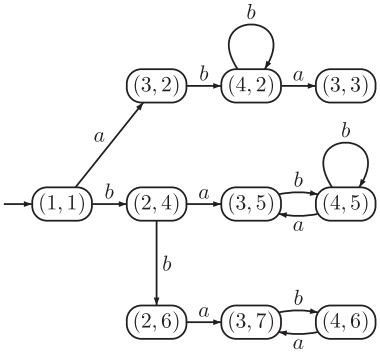
$$\mathcal{R}^P\cdot\delta((q,q'),a) = \begin{cases} (\mathcal{R}\cdot\delta(q,a), \mathcal{M}\cdot\delta(q',a)) & \text{if } \mathcal{R}\cdot\delta(q,a) \text{ and } \mathcal{M}\cdot\delta(q',a) \text{ are defined} \\ \text{undefined otherwise} & \end{cases}$$

- $\mathcal{R}^P\cdot C = \cup_{(L,U)\in\mathcal{R}\cdot C}\{(L \times \mathcal{M}\cdot Q, U \times \mathcal{M}\cdot Q)\}$

The automaton built for the example using the property in Fig. 1 and the program model presented in Fig. 2 is given in Fig. 3.

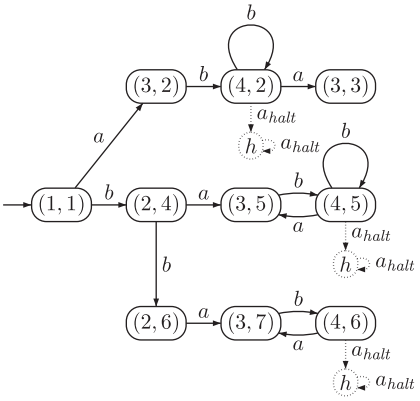
Since $\mathcal{R}^P$ accepts the intersection of the languages accepted by the automaton $\mathcal{R}$ and $\mathcal{M}$, it would seem an ideal abstraction from which to build the instrumented program. However, there is no known way to transform such an automaton into a program. Indeed, since the acceptance condition of the Rabin automaton is built around the notion of infinite traces reaching some states infinitely often, a dynamic monitoring system built from such an automaton with no help provided by a prior static analysis, may never be able to determine if a given execution is valid or not.

Instead, a deterministic automaton, $\mathcal{T} = (\mathcal{T}\cdot Q, \mathcal{T}\cdot q_0, \mathcal{T}\cdot\delta)$ is extracted from the Rabin automaton $\mathcal{R}^P$. This automaton is the labeled transition system which is returned. It forms in turn the basis of the instrumented program being constructed. The instrumented program is expected to work as a program monitored by a truncation automaton meaning that its model $\mathcal{T}$ has to satisfy the following conditions: (1) $\mathcal{T}$ emits each execution of $\mathcal{M}$ satisfying the security property without any modification, (2) for each execution that does not satisfy the property, $\mathcal{T}$ safely halts it after producing a valid partial



$$C = \{(\ \{(3,2),(3,3),(3,5),(3,7)\},\ \{(4,2),(4,5),(4,6)\}\ )\}$$

Fig. 3 – Example – Rabin automaton $\mathcal{R}^P$.



$$C = \{(\{(3,2),(3,3),(3,5),(3,7)\},\ \{(4,2),(4,5),(4,6)\}),(\emptyset,\{h\})\}$$

Fig. 4 – Transformed product automaton.

execution, and (3) $\mathcal{T}$ does not emit anything else apart those executions described in (1) and (2).

The next step toward this goal is to apply a transformation that allows $\mathcal{R}^P$ to accept partial executions of $\mathcal{M}$ which satisfy the property. Indeed, all finite initial paths in $\mathcal{R}^P$ represent partial executions of $\mathcal{M}$, but only some of them satisfy the security property. To this end, a transition labeled a_{halt} to a new state h is added to every state in R^P where the execution could be aborted after producing a partial execution satisfying the property, i.e. a state (q_1,q_2) for which $\mathcal{R}\cdot\delta(q_1, a_{\text{end}})$ is defined. The state h is made admissible by adding the transition (h,a_{halt},h) to the set of transitions and the pair ($\emptyset$,{h}) to the acceptance set. Care must be taken in choosing h and a_{halt} such that $h \notin \mathcal{R}\cdot Q \cup \mathcal{M}\cdot Q$ and $a_{\text{halt}} \notin \mathcal{A}$ the alphabet of actions.

Let $\mathcal{R}^T$ stand for this updated version of $\mathcal{R}^P$, built as follows:

- $\mathcal{R}^T\cdot Q = \mathcal{R}^P\cdot Q \cup \{h\}$
- $\mathcal{R}^T\cdot q_0 = \mathcal{R}^P\cdot q_0$
- $\mathcal{R}^T\cdot\delta = \mathcal{R}^P\cdot\delta \cup \{(q, a_{\text{halt}}, h) | \mathcal{R}^P\cdot\delta(q, a_{\text{end}})$ is defined$\} \cup \{(h, a_{\text{halt}}, h)\}$
- $\mathcal{R}^T\cdot C = \mathcal{R}^P\cdot C \cup \{(\emptyset, \{h\})\}$

After this transformation the example product automaton becomes the automaton depicted in Fig. 4. The halt state h has been duplicated three times in order to avoid cross edging.

The language recognized by $\mathcal{R}^T$ is

$$\mathcal{L}_{\mathcal{R}^T} = (\mathcal{L}_{\mathcal{R}} \cap \mathcal{L}_{\mathcal{M}}) \cup \{\tau; (a_{\text{halt}})^\omega | (\tau \in \mathcal{A}^*) \wedge (\exists \sigma \in \mathcal{L}_{\mathcal{M}} : \tau \preccurlyeq \sigma) \wedge (\tau; (a_{\text{end}})^\omega \in \mathcal{L}_{\mathcal{R}})\}.$$

4.3.2. *Extracting a model of the instrumented program*

The next phase consists in extracting, if possible, a labeled transition system $\mathcal{T} = (Q, q_0, \delta)$, from the Rabin automaton $\mathcal{R}^T$. This automaton is expected to behave as the original program monitored by a truncation automaton.

To understand the need for this step, first note that the acceptance condition of a Rabin automaton could not be checked dynamically due to its infinite nature. Should an instrumented program be built directly from $\mathcal{R}^T$, by ignoring its acceptance condition, and treating it like a simple LTS, the resulting program would still generate all traces of $\mathcal{M}$ that verify the property $\widehat{\mathcal{P}}$ but it would also generate the invalid sequences of $\mathcal{M}$ representing labels of infinite paths in $\mathcal{R}^T$ trapped in non admissible cycles. In other words, the enforcement of the property would be transparent but not sound.

In order to generate $\mathcal{T}$, $\mathcal{R}^T$ is pruned of some of its states and transitions, eliminating inadmissible cycles while taking care to preserve the ability to generate all the valid sequences of $\mathcal{L}_{\mathcal{M}}$. Furthermore, it is essential ascertain that $\mathcal{T}$ aborts the execution of every sequence of $\mathcal{L}_{\mathcal{M}}$ not satisfying $\widehat{\mathcal{P}}$ and that $\mathcal{T}$ generates only executions satisfying $\widehat{\mathcal{P}}$.

The correctness requirements of the approach can now be restated. In the formulation of these requirements, the actions a_{end} and a_{halt} are ignored, as they merely model the end of a finite sequence.

$$(\forall \sigma \in \mathcal{L}_{\mathcal{M}} | : (\exists \tau \in \mathcal{L}_{\mathcal{T}} | : ((\tau = \sigma) \vee (\tau \preccurlyeq \sigma)) \wedge \widehat{\mathcal{P}}(\tau) \wedge (\widehat{\mathcal{P}}(\sigma) \Rightarrow (\tau = \sigma)))) \tag{4.1}$$

$$\forall \tau \in \mathcal{L}_{\mathcal{T}} | : (\exists \sigma \in \mathcal{L}_{\mathcal{M}} | : ((\tau = \sigma) \vee (\tau \preccurlyeq \sigma))) \wedge \widehat{\mathcal{P}}(\tau) \tag{4.2}$$

Note that the requirements (4.1) and (4.2) are not only sufficient to ensure the respect of soundness and transparency requirements introduced at the beginning of Section 4 following Erlingsson (2004), Hamlen et al. (2006) and Ligatti et al. (2005), but also that of a more restrictive requirement. Indeed, requirement (4.1) also states that the mechanism is a truncation mechanism. It ensures the compliance to the security property by aborting the execution before a security violation occurs whenever this is needed. It follows that for any invalid sequence present in the original model, the instrumented program outputs a valid prefix of that sequence.

The enforcement mechanism is not allowed to generate sequences that are not related to sequences in $\mathcal{L}_{\mathcal{M}}$ either by equality or prefix relation. Furthermore these sequences must satisfy $\widehat{\mathcal{P}}$. This is stated in requirement (4.2).

Requirements (4.1) and (4.2) give the guidelines for constructing $\mathcal{T}$ from $\mathcal{R}^T$. The transformations that are performed on $\mathcal{R}^T$ to ensure meeting these requirements are elaborated around the following intuition. The automaton $\mathcal{R}^T$ has to be pruned so as to ensure that it represents a safety property even though $\mathcal{R}$ is not. Note that this is not possible in the general case without violating the requirements. The idea is that

admissible cycles are visited infinitely often by executions satisfying $\widehat{\mathcal{P}}$ and must thus be included in $\mathcal{T}$. Likewise, any other state or transition that can reach an admissible cycle may be part of such an execution and must be included. On the other hand, inadmissible cycles cannot be included in $\mathcal{T}$ as the property is violated by any trace that goes through such a cycle infinitely often. In some cases their elimination cannot occur without the loss of transparency and the approach fails, returning **error**. The underlying idea of the subsequent manipulation is thus to check whether $\mathcal{R}^T$ can be trimmed by removing bad cycles but without also removing the states and transitions required to ensure transparency.

The following steps show how the trim procedure is performed.

The next step is to determine the strongly connected components (*scc*) in the graph representing $\mathcal{R}^T$ using Tarjan's algorithm (Tarjan, 1972). Each *scc* is then examined and mark it as containing either only admissible cycles, only inadmissible cycles, both types of cycles, or no cycles (in the trivial case).

The next step is to construct the quotient graph of $\mathcal{R}^T$ in which each node represents a *scc* and an edge connecting two *scc* c_1 and c_2 indicates that there exists a state q_1 in *scc* c_1 and a state q_2 in *scc* c_2 and an action a such that $\mathcal{R}^T \cdot \delta(q_1, a) = q_2$. We assume, without loss of generality, that all the *scc* states are accessible from the initial node, the *scc* containing q_0.

The nodes of the quotient graph $\mathcal{R}^T$ are then visited in reverse topological ordering in order to determine, for each one, whether it should be kept intact, altered or removed.

In what follows the *scc* containing the halting state h is referred to as H.

A *scc* with no cycle at all is removed with its incident edges if it cannot reach another *scc*. In Fig. 4 the *scc* consisting of the state (3,3) would thus be eliminated.

A *scc* containing only admissible cycles should be kept, since all the executions reaching it satisfy $\widehat{\mathcal{P}}$. Eliminating it would prevent the enforcement mechanism from being transparent. In the example in Fig. 4 the *scc* consisting of the single state (4,2) has only admissible cycles and should be kept.

A *scc* containing only non admissible cycles can be removed if it cannot reach another *scc* with only admissible cycles. Otherwise, the algorithm is generally forced to return **error**. However, in some cases, it is possible to either break the inadmissible cycles or prevent them from reaching H by removing some transitions and keeping the remainder of the *scc*. This occurs when the only successor, having admissible cycles, of this *scc* is H. In the example, the *scc* containing the states (3,7) and (4,6) has only non admissible cycles and H is its only successor. This *scc* can be eliminated and halting with **error** at this point. Yet, if eliminating the transition ((4,6),a,(3,7)) would break the inadmissible cycle, the algorithm does so, keeping the rest of the *scc*.

A transition can only be removed if its origin has h as immediate successor. This is because, should the instrumented program attempt to perform the action that corresponds to this transition, its execution would be aborted. However, a partial execution only satisfies the property if it ends in a state that has h as an immediate successor.

A *scc* containing admissible and non admissible cycles may cause good or bad behavior. Actually, an execution reaching this *scc* may be trapped in an inadmissible cycle for ever or may leave it to reach an admissible cycle thus satisfying the property $\widehat{\mathcal{P}}$. There are no means to dynamically check whether the execution is going to leave a cycle or not. Thus, in this case the algorithm must abort with **error**. In the example given in Fig. 4 the *scc* consisting of the two states (3,5) and (4,5) have one admissible cycle, $\langle(4,5),(4,5)\rangle$ and one inadmissible cycle $\langle(3,5),(4,5),(3,5)\rangle$. This last cycle is visited if the invalid sequence $(ba)^\omega$ is being generated. Note that the automaton accepts an infinite number of valid traces of the form ba $(ba)^*b^\omega$, and that no truncation automaton can both accept these traces and reject the invalid trace described above. Hence the algorithm aborts with **error** in such cases.

After removing all the *scc* with inadmissible cycles and provided the algorithm did not abort, it is certain that an instrumented program built from $\mathcal{T}$ would not contain any infinite-length execution which does not respect the security property. It is still necessary to verify that whenever the execution is halted, the partial sequence emitted satisfies $\widehat{\mathcal{P}}$.

The last step is to check whether the eliminated states and transitions could not allow invalid partial executions to be emitted. This verification is based on the following observation: if a removed transition has an origin state that is not an immediate predecessor of h this would then allow to emit a partial execution that does not satisfy $\widehat{\mathcal{P}}$. Hence, the verification merely consists in checking whether any transitions from states that are not immediate predecessors of h have been removed; if such is the case, the algorithm aborts with **error**. More precisely, for a state $q = (q_1,q_2)$ in $\mathcal{T}$, if it is possible to perform actions from q_2 in $\mathcal{M}$ that are not possible from q, then q must have h as immediate successor, otherwise there are no other option than to terminate the algorithm without returning a suitable LTS and with an error message.

Transitions of the form (h,a_{halt},h) and (q,a_{end},q), where $q \in \mathcal{R}^T \cdot Q$ can also be removed.

4.4. *Additional example*

Throughout this section, each step of the approach has been illustrated with an example that was carefully crafted to highlight some of the behavior the algorithm may encounter when in-lining a monitor into a target program. As the behavior of this program lead to the rejection by the transformation algorithm, we were unable to show how, using this example, a program can successfully be instrumented to ensure the respect of a non-safety property. We thus introduce in this section another example in which the approach succeeds and returns a model of an instrumented program that verifies the property. The most striking feature of this example is the fact that the property being enforced is not a safety property and, as such, cannot possibly be enforced under existing implementations on formal in-line monitoring frameworks.

This security property is modeled by the automaton in Fig. 5, over the alphabet $\mathcal{A} = \{\text{open}, \text{close}, \text{void}\}$. This automaton captures a plausible security requirement for a program that accesses a database, namely that:

- The program can open no more that one connection to the database at any given time.

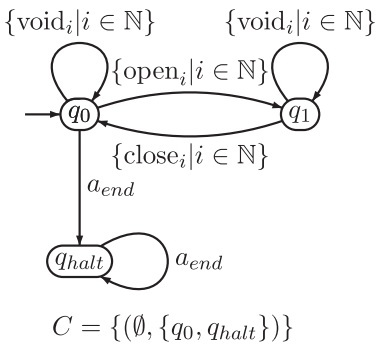


Fig. 5 – Example 2, Rabin automaton.

- The program only closes a connection to the database if it has already been opened.
- Any connection that is opened is eventually closed. This last requirement adds a liveness component to the desired security property.

Note that the first two actions from $\mathcal{A}$ model the operation of opening and closing the database, while the last is used as a stand in for other program actions that have no bearing on the satisfaction of the security property.

Fig. 6 shows an LTS which approximates the behavior of the target program. This program could be a remote agent who accesses a database, performs some computations locally and returns a result. The subscripts added to the atomic actions serve only to avoid the presence of non-determinism in the model (each action is associated with a distinct program instruction), and has no bearing on the satisfaction of the security predicate.

The transformed product automaton $\mathcal{R}^T$ of Example 2 is depicted in Fig. 7.

In Fig. 7, the strongly connected components are shown and annotated with either *A* meaning all the component's cycles are admissible, *NC* meaning the component has no cycles and *N* meaning all the cycles of the component are inadmissible.

This section omits the intermediate steps performed by the algorithm, since they have already been discussed throughout the last section using the preceding example. More relevant is the result returned by the algorithm in this case, namely an LTS that could be transformed into a provably secure program. This LTS is presented in Fig. 8.

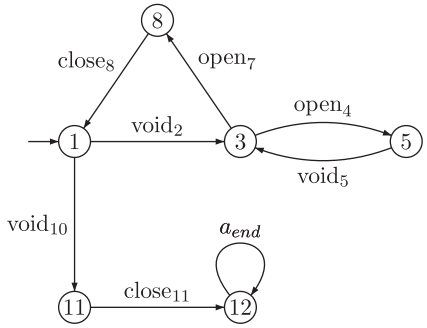


Fig. 6 – Example 2, LTS.

5. Mechanism's enforcement power

In this section shows that nonuniform enforcement mechanisms, which occur when the set of possible executions Σ is a subset if A^ω, are more powerful than uniform enforcers, i.e. those for which $\Sigma = A^\omega$, in the sense that they are able to enforce a larger class of security properties. This demonstration will reveal that monitors that are tailored to specific programs may be able to enforce a wide set of properties and argues for the use of static analysis in conjunction with monitoring.

Let us begin with a more formal definition of the concepts discussed in the previous sections, following the notations adopted in Bauer et al. (2002) and Ligatti et al. (2005a). The behavior of a security automaton S is specified by judgments of the form $(q,\sigma)\rightarrow_S(q',\sigma)$ where q is the current state of the automaton; q' is the attempted execution; σ is the state the automaton reach after one execution step; σ' is the remaining execution trace to be performed; and τ is the execution trace consisting of one action at most that is emitted by the security automaton after one step.

The execution of the security automaton is generalized with the multi-step judgments defined through reflexivity and transitivity rules as follows.

Definition 5.1. (*Multi-step semantics, from Bauer et al. (2002)*) Let S be a security automaton. The multi-step relation $(q,\sigma)\overset{\tau}{\Longrightarrow}s(q',\sigma')$ is inductively defined as follows.

For all $q,q',q''\in Q$, $\sigma,\sigma',\sigma''\in\mathcal{A}^\infty$ and $\tau,\tau'\in\mathcal{A}^*$

$$(q,\sigma)\overset{\epsilon}{\underset{S}{\Longrightarrow}}(q,\sigma) \tag{5.1}$$

$$\text{if } (q,\sigma)\overset{\tau}{\underset{S}{\Longrightarrow}}(q'',\sigma'') \text{ and } (q'',\sigma'')\overset{\tau'}{\rightarrow}_S(q\prime,\sigma\prime) \text{ then } (q,\sigma)\overset{\tau;\tau\prime}{\underset{S}{\Longrightarrow}}(q',\sigma') \tag{5.2}$$

A formal definition of what a security enforcement mechanism is can now be given. Intuitively, security enforcement mechanisms can be thought of as sequence transformers, automata that take a program's actions sequence as input, and output a new sequence of actions that respects the security property. This intuition is formalized as follows:

Definition 5.2. (*Transformation, from Bauer et al. (2002)*) A security automaton $S=(Q,q_0,\delta)$ transforms an execution trace $\sigma\in\mathcal{A}^\infty$ into an execution $\tau\in\mathcal{A}^\infty$, noted $(q_0,\sigma)\Downarrow_S\tau$, if and only if

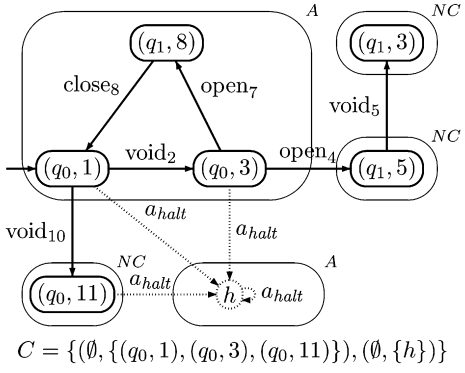


Fig. 7 – Example 2, transformed product automaton.

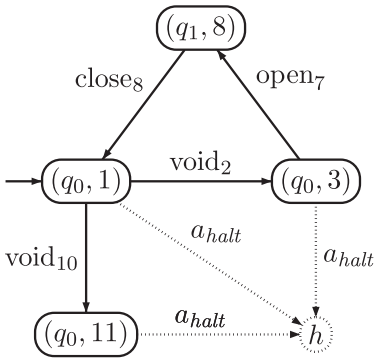


Fig. 8 – Example 2, truncation automaton.

$$\forall q \in Q, \sigma' \in \mathcal{A}^\infty, \tau' \in \mathcal{A}^* \mid : \left((q_0\sigma) \overset{\tau'}{\Rightarrow} (q', \sigma')\right) \Rightarrow \tau' \preccurlyeq \tau \tag{5.3}$$

$$\forall \tau\prime \preccurlyeq \tau \mid : \exists q' \in Q, \sigma' \in \mathcal{A}^\infty \mid : (q_0, \sigma) \overset{\tau'}{\underset{S}{\Rightarrow}} (q', \sigma') \tag{5.4}$$

The definition of effective$_{\cong}$ enforcement can now be formally stated.

Definition 5.3. (*effective$^{\Sigma}_{\cong}$ Enforcement, from Bauer et al. (2002)*) Let $\Sigma \subseteq \mathcal{A}^\infty$ be a set of execution traces. A security automaton $S = (Q, q_0, \delta)$ enforces effectively$_{\cong}$ a security property $\widehat{\mathcal{P}}$ for Σ if and only if for all input trace $\sigma \in \Sigma$ there exists an output trace $\tau \in \mathcal{A}^\infty$ such that

$$(q_0, \sigma) \Downarrow_S \tau \tag{5.5}$$

$$\widehat{\mathcal{P}}(\tau) \tag{5.6}$$

$$\widehat{\mathcal{P}}(\sigma) \Rightarrow \sigma \cong \tau \tag{5.7}$$

Informally, a security automaton effectively$_{\cong}$ enforces a property for Σ *iff* for each execution trace $\sigma \in \Sigma$, it outputs a trace τ such that τ is valid, with respect to the property, and if the input trace σ is itself valid then $\sigma \cong \tau$.

Definition 5.4. ($\mathcal{S}^{\Sigma}_{\cong}$*-enforceable*) Let $\Sigma \subseteq \mathcal{A}^\infty$ be a set of execution traces and $\mathcal{S}$ be a class of security automata. The class $\mathcal{S}^{\Sigma}_{\cong}$-enforceable is the set of security properties such that for each property in this set, there exists a security automaton $S \in \mathcal{S}$ that effectively$_{\cong}$ enforces this property for the traces in Σ.

The algorithm is built around the idea, first suggested by Bauer et al. (2002) and Ligatti et al. (2005a), that the set of properties enforceable by a monitor could sometimes be extended if the monitor has some knowledge of the program's possible behavior and thus can rule out some executions as impossible.

This idea can now be stated more formally.

Theorem 5.5. *Let $\mathcal{S}$ be a class of security automata and let $\Sigma^\natural, \Sigma^\sharp \subseteq \mathcal{A}^\infty$ be two sets of execution traces $\Sigma^\natural \subseteq \Sigma^\sharp$ then*

$$\mathcal{S}^{\Sigma^\natural}_{\cong} - \text{enforceable} \subseteq \mathcal{S}^{\Sigma^\sharp}_{\cong} - \text{enforceable} \tag{5.8}$$

The proof is quite straightforward, and based upon the intuition that a security mechanism possessing certain knowledge about its target may discard it, and then behave as an enforcement mechanisms lacking this knowledge.

Proof 1. Let $\Sigma^\natural, \Sigma^\sharp \subseteq \mathcal{A}^\infty$ be two sets of execution traces such that $\Sigma^\natural \subseteq \Sigma^\sharp$ and $\widehat{\mathcal{P}}$ be a $\mathcal{S}^{\Sigma^\sharp}_{\cong}$ – enforceable security property.

$$\begin{aligned}
&\widehat{\mathcal{P}} \in \mathcal{S}^{\Sigma^\sharp}_{\cong} - \text{enforceable} \\
&\Leftrightarrow \langle \text{Definitions 5.3 and 5.4} \rangle \\
&\left(\exists S = (Q, q_0, \delta) \in \mathcal{S} \,\middle|\, : \left(\forall \sigma \in \Sigma^\sharp \,\middle|\, : \left(\exists \tau \in \mathcal{A}^\infty \,\middle|\, : \begin{array}{c} (q_0\sigma) \Downarrow_S \tau \\ \wedge \quad \widehat{\mathcal{P}}(\tau) \\ \wedge \quad (\widehat{\mathcal{P}}(\sigma) \Rightarrow \sigma \cong \tau) \end{array}\right)\right)\right) \\
&\Leftrightarrow \langle \text{Domain weakening } (\Sigma^\natural \subseteq \Sigma^\sharp) \rangle \\
&\left(\exists S = (Q, q_0, \delta) \in \mathcal{S} \,\middle|\, : \left(\forall \sigma \in \Sigma^\natural \,\middle|\, : \left(\exists \tau \in \mathcal{A}^\infty \,\middle|\, : \begin{array}{c} (q_0\sigma) \Downarrow_S \tau \\ \wedge \quad \widehat{\mathcal{P}}(\tau) \\ \wedge \quad (\widehat{\mathcal{P}}(\sigma) \Rightarrow \sigma \cong \tau) \end{array}\right)\right)\right) \\
&\Leftrightarrow \langle \text{Definitions 5.3 and 5.4} \rangle \\
&\widehat{\mathcal{P}} \in \mathcal{S}^{\Sigma^\natural}_{\cong} - \text{enforceable}
\end{aligned}$$

Corollary 5.6. *Let $\mathcal{S}$ be a class of security automaton. For all execution trace set $\Sigma \subseteq \mathcal{A}^\infty$*

$$\mathcal{S}^{\mathcal{A}^\infty}_{\cong} - \text{enforceable} \subseteq \mathcal{S}^{\Sigma}_{\cong} - \text{enforceable} \tag{5.9}$$

Corollary 5.6 indicates that any security property that is effectively $\cong$ enforceable by a security automaton in a uniform context ($\Sigma = \mathcal{A}^\infty$) is also enforceable in the nonuniform context ($\Sigma \neq \mathcal{A}^\infty$). It follows that that the approach presented in this paper is at least as powerful as those previously suggested in the literature.

It would be interesting to prove that for all security automaton classes, $\mathcal{S}$ and for all equivalence relations $\cong$, $\mathcal{S}^{\mathcal{A}^\infty}_{\cong} - \text{enforceable} \subset \mathcal{S}^{\Sigma}_{\cong} - \text{enforceable}$.

This is unfortunately not the case, as there exists at least one class of security automaton (ex. $\mathcal{S} = \emptyset$), and one equivalence relation (ex. $\tau \cong \sigma \; \forall \tau, \sigma \in \mathcal{A}^\infty$) such that $\mathcal{S}^{\mathcal{A}^\infty}_{\cong}$ –enforceable $= \mathcal{S}^{\Sigma}_{\cong}$ – enforceable for all set of traces $\Sigma \subseteq \mathcal{A}^\infty$.

However in this approach focuses both on a specific class of security automata and on a specific equivalence relation. In this particular case, the set of policies enforceable in a nonuniform context is strictly greater than the one that is enforceable in the uniform context.

The monitors used in this paper are truncation automata, first described in Schneider (2000). These are monitors which, when presented with a potentially invalid sequence, have no option but to abort the execution.

Definition 5.7. (*Truncation Automaton, from Schneider (2000)*) A truncation automaton is a security automaton where $\delta : Q \times \mathcal{A} \rightarrow Q \cup \{\text{halt}\}$ and $\text{halt} \notin Q$.

The following theorem gives a characterization of the properties that can be effectively $\cong$ enforced in a nonuniform context.

Theorem 5.8. *Let $\Sigma \subseteq \mathcal{A}^\infty$ be a set of execution sequences. A property $\widehat{\mathcal{P}}$ is $T^{\Sigma}_{\cong}$ iff there exists a decidable predicate D over $\mathcal{A}^*$ such that*

$$\left(\forall \sigma \in \Sigma | \neg \widehat{\mathcal{P}}(\sigma) : \left(\exists \sigma\prime \preccurlyeq \sigma | : D(\sigma\prime)\right)\right) \tag{5.10}$$

$$\left(\forall \tau; a \in \mathcal{A}^* \middle| D(\tau; a) \widehat{\mathcal{P}}(\tau) \wedge \left(\forall \sigma \in \Sigma \middle| \sigma \succcurlyeq \tau; a \wedge \widehat{\mathcal{P}}(\sigma) : \sigma \cong \tau\right)\right) \tag{5.11}$$

$$\neg D(\epsilon) \tag{5.12}$$

Proof 2. (*if direction*) Let $\Sigma \subseteq \mathcal{A}^\infty$ be a set of execution sequences, let $\widehat{\mathcal{P}}$ be a property, $\cong$ an equivalence relation over the execution sequences and D be a decidable predicate over $\mathcal{A}^*$ which satisfies the conditions (5.10)–(5.12). The truncation automaton T effectively $\cong$ enforces $\widehat{\mathcal{P}}$ over Σ.

Let $T = (Q, q_0, \delta)$ be a truncation automaton such that

- $Q = \mathcal{A}^*$;
- $q_0 = \epsilon$;
- For all $\sigma \in \mathcal{A}^*$, $a \in \mathcal{A}$

$$\delta(\sigma, a) = \begin{cases} \sigma; a & \text{if } \neg D(\sigma; a) \\ \text{halt} & \text{else} \end{cases}$$

Since $\mathcal{A}^*$ is a countably infinite set, so is Q. Furthermore, δ is fully defined and computable since D is decidable.

The automaton T satisfies the following invariant $I(q)$: for any reachable state $q = \sigma$, the execution sequence σ has been emitted so far, $(q_0, \sigma) \Downarrow_T \sigma$ and $(\forall \sigma\prime \preccurlyeq \sigma | : \neg D(\sigma\prime))$. Informally, if the automaton T is in state $q = \sigma$, it did not halt before reaching this state and has emitted exactly the execution sequence σ. T obviously satisfies the $I(q_0)$ since $q_0 = \varepsilon$, $(q_0, \epsilon) \Downarrow_T \epsilon$ and $\neg D(\epsilon)$. An induction on the length of the execution trace shows that $I(q)$ is satisfied for any reachable state $q = \sigma$.

Let $\sigma \in \Sigma$ be an input sequence. The proof that T effectively$_{\cong}$ enforces $\widehat{\mathcal{P}}$ over Σ proceeds by showing that T satisfies conditions (5.5)–(5.7) for σ.

- if $\neg \widehat{\mathcal{P}}(\sigma)$

  By condition (5.10), $(\exists \sigma\prime \preccurlyeq \sigma | : D(\sigma\prime))$. Thus, by invariant I, T must halt if its input sequence is σ, and must do so before reaching the state σ'. Let τ be the last state which T reaches when its input is σ. By I and Definition 5.2, $(q_0, \sigma) \Downarrow_T \tau$. Thus condition (5.5) is satisfied. Let $a \in \mathcal{A}$ be an action such that $\tau; a \preccurlyeq \sigma$. Since τ is the last state T reaches when σ is input and by the definition of T $\delta(\tau, a) = \text{halt}$ and thus that $D(\tau; a)$. Furthermore, by condition (5.11) $\widehat{\mathcal{P}}(\tau)$. It follows that condition (5.6) is satisfied. Since $\neg \widehat{\mathcal{P}}(\sigma)$, condition (5.7) is satisfied.
- if $\widehat{\mathcal{P}}(\sigma)$
  - if T does not halt on input σ

    then T emits all and every prefix of σ. By Definition 5.2, $(q_0, \sigma) \Downarrow_T \sigma$. Thus, condition (5.5) is satisfied. Since $\widehat{\mathcal{P}}(\sigma) \cong$ is reflexive, conditions (5.6) and (5.7) are also satisfied.
  - if T halts on input σ

    Let τ be the last state T reaches when its input is σ. By I and Definition 5.2 then $(q_0, \sigma) \Downarrow_T \tau$. Thus, la condition (5.5) is satisfied. Let $a \in \mathcal{A}$ be an action such that $\tau; a \preccurlyeq \sigma$. Since τ is the last state T reaches when σ is input and by the definition of T it follows that $\delta(\tau, a) = \text{halt}$ and thus that $D(\tau; a)$. By condition (5.11), $\widehat{\mathcal{P}}(\tau)$ and since $\widehat{\mathcal{P}}(\sigma)$, $\sigma \cong \tau$. Thus conditions (5.6) and (5.7) are satisfied.

Only if direction Let $\Sigma \subseteq \mathcal{A}^\infty$ be a set of execution sequences and let $\widehat{\mathcal{P}}$ be a property, $\cong$ an equivalence relation over the execution sequences and $T = (Q, q_0, \delta)$ be a truncation automaton which *effectively*$_{\cong}$ enforces $\widehat{\mathcal{P}}$ over Σ.

The proofs follows from the construction of a decidable D over $\mathcal{A}^*$ such that the conditions (5.10)–(5.12) are satisfied.

This predicate is built in the following manner:

- $D(\epsilon)$ is false.
- For all $\sigma \in \mathcal{A}^*$, $a \in \mathcal{A}$ then $D(\sigma; a)$ is true iff automaton T emits exactly σ when the input sequence is $\sigma; a$.

Since δ is fully defined and computable, and D is only defined over $\mathcal{A}^*$ then D is also fully defined and computable.

The definition of D implies that condition (5.12) is satisfied.

Furthermore, since T effectively$_{\cong}$ enforces $\widehat{\mathcal{P}}$ over Σ, if it outputs σ the input sequence is $\sigma; a$, then, by condition (5.6), $\widehat{\mathcal{P}}(\sigma)$. Also, by condition (5.7) the fact that T halts before emitting any sequence $\tau \in \Sigma$ such that $\tau \succcurlyeq \sigma; a$, condition (5.11) is satisfied.

Finally, let $\sigma \in \Sigma$ be an input trace such that $\neg \widehat{\mathcal{P}}(\sigma)$ and assume that $(\forall \sigma\prime \preccurlyeq \sigma | : \neg D(\sigma\prime))$ as to obtain a contradiction. By the definition of D, automaton T cannot halt on any prefix of σ, and must thus necessarily output all its prefixes. Yet by Definition 5.2, $(q_0, \sigma) \Downarrow_T \sigma$.

This is a contradiction since T cannot *effectively*$_{\cong}$ enforce $\widehat{\mathcal{P}}$ over Σ if there exists an input trace $\sigma \in \Sigma$ such that $(q_0, \sigma) \Downarrow_T \sigma$ and $\neg \widehat{\mathcal{P}}(\sigma)$. It follows that condition (5.10) is satisfied.

In this study it is also restricted to the use of syntactic equivalence ($=$) as the equivalence relation between valid traces. We thus give a characterization of the security properties effectively$_{=}$ by an truncation automaton, in a nonuniform context.

Corollary 5.9. *Let $\Sigma \subseteq \mathcal{A}^\infty$ be a set of execution sequences. A property $\widehat{\mathcal{P}}$ is $T^{\Sigma}_{=}$ – enforceable iff there exists a decidable predicate D over $\mathcal{A}^*$ such that*

$$\left(\forall \sigma \in \Sigma | \neg \widehat{\mathcal{P}}(\sigma) : \left(\exists \sigma\prime \preccurlyeq \sigma | : D(\sigma\prime)\right)\right) \tag{5.13}$$

$$\left(\forall \tau; a \in \mathcal{A}^* \middle| D(\tau; a) : \widehat{\mathcal{P}}(\tau) \wedge \left(\forall \sigma \in \Sigma \middle| \sigma \succcurlyeq \tau; a : \neg \widehat{\mathcal{P}}(\sigma)\right)\right) \tag{5.14}$$

$$\neg D(\epsilon) \tag{5.15}$$

Proof 3. Let $\Sigma \subseteq \mathcal{A}^\infty$ be a set of execution sequences and let $\widehat{\mathcal{P}}$ be a security property such that $\widehat{\mathcal{P}} \in T^{\Sigma}_{=}$-enforceable.

$\widehat{\mathcal{P}} \in T^{\Sigma}_{=}$ – enforceable

$\Leftrightarrow \langle$Theorem 5.8$\rangle$

$$\left(\exists D \middle| \begin{array}{ll} & \left(\forall \sigma \in \Sigma | \neg \widehat{\mathcal{P}}(\sigma) : \left(\exists \sigma' \preccurlyeq \sigma | : D(\sigma')\right)\right) \\ : \wedge & \left(\forall \tau; a \in \mathcal{A}^* | D(\tau; a) : \widehat{\mathcal{P}}(\tau) \wedge \left(\forall \sigma \in \Sigma | \sigma \succcurlyeq \tau; a \wedge \widehat{\mathcal{P}}(\sigma) : \sigma = \tau\right)\right) \\ \wedge & \neg D(\epsilon) \end{array}\right)$$

$\Leftrightarrow \langle$Transfer$\rangle$

$$\left(\exists D \middle| \begin{array}{ll} & \left(\forall \sigma \in \Sigma | \neg \widehat{\mathcal{P}}(\sigma) : \left(\exists \sigma' \preccurlyeq \sigma | : D(\sigma')\right)\right) \\ : \wedge & \left(\forall \tau; a \in \mathcal{A}^* | D(\tau; a) : \widehat{\mathcal{P}}(\tau) \wedge \left(\forall \sigma \in \Sigma | \sigma \succcurlyeq \tau; a : \widehat{\mathcal{P}}(\sigma) \Rightarrow \sigma = \tau\right)\right) \\ \wedge & \neg D(\epsilon) \end{array}\right)$$

$\Leftrightarrow \langle \sigma \succcurlyeq \tau; a \Rightarrow \sigma \succ \tau \Rightarrow \sigma \neq \tau \rangle$ and $\langle p \Rightarrow \text{false}) \Leftrightarrow (\neg p) \rangle$

$$\left(\exists D \middle| \begin{array}{ll} & \left(\forall \sigma \in \Sigma | \neg \widehat{\mathcal{P}}(\sigma) : \left(\exists \sigma' \preccurlyeq \sigma | : D(\sigma')\right)\right) \\ : \wedge & \left(\forall \tau; a \in \mathcal{A}^* | D(\tau; a) : \widehat{\mathcal{P}}(\tau) \wedge \left(\forall \sigma \in \Sigma | \sigma \succcurlyeq \tau; a : \neg \widehat{\mathcal{P}}(\sigma)\right)\right) \\ \wedge & \neg D(\epsilon) \end{array}\right)$$

We can now show that the enforcement power of the truncation automaton is strictly greater in the nonuniform context than in the uniform context.

Theorem 5.10. *For all set of traces $\Sigma \subset \mathcal{A}^\infty$*

$$T^{\mathcal{A}^\infty}_{=} \text{ – enforceable} \subset T^{\Sigma}_{=} \text{ – enforceable} \tag{5.16}$$

The proof is based on the following observations. First, it has been shown in Schneider (2000) and Bauer et al. (2002) that a property is $T_{=}^{\mathcal{A}^\infty}$-enforceable *iff* it is a safety property. Second. Let $\widehat{\mathcal{P}}$ be a security property, $\widehat{\mathcal{P}}$ is trivially enforceable on Σ *iff* for every sequence $\sigma \in \Sigma, \widehat{\mathcal{P}}(\sigma)$.

Let $T = (Q, q_0, \delta)$ be a truncation automaton such that for every sequence $\sigma \in \mathcal{A}^\infty\, (q_0, \sigma) \Downarrow_T \sigma$. It is easy to see that this automaton trivially enforces over the set Σ any trivially enforceable property over this same set. The remainder of this proof consists in showing that there exist some properties which are trivially enforceable over Σ but are not reasonable, safety and decidable.

Let $\upsilon \in \mathcal{A}^\infty$ be an execution sequence such that $\upsilon \notin \Sigma$ and let $\widehat{\mathcal{P}}$ be the security property stating that for any sequence $\sigma \in \mathcal{A}^\infty$,

$$\widehat{\mathcal{P}}(\sigma) \Leftrightarrow (\sigma \neq \upsilon)$$

This property is $T_{=}^{\Sigma}$-enforceable, and the following automaton, $T = (Q, q_0, \delta)$ with $\delta(q_0, a) = q_0$ for all actions $a \in \mathcal{A}$ effectively enforces it over $\mathcal{A}$.

That $\widehat{\mathcal{P}} \notin T_{=}^{\mathcal{A}^\infty}$ – enforceable is proved by contradiction. Let $\widehat{\mathcal{P}} \in T_{=}^{\mathcal{A}^\infty}$ – enforceable, by definition, there exists a predicate D over $\mathcal{A}^*$ such that conditions (5.13)–(5.15) are satisfied.

$\widehat{\mathcal{P}} \in T_{=}^{\mathcal{A}^\infty}$ – enforceable
$\Leftrightarrow$ ⟨corollary (5.9) and definition of $\widehat{\mathcal{P}}$ ⟩

$$\left(\exists D \left| \begin{array}{ll} & (\forall \sigma \in \mathcal{A}^\infty | \sigma = \upsilon : (\exists \sigma' \preccurlyeq \sigma | : D(\sigma'))) \\ : \wedge & (\forall \tau; a \in \mathcal{A}^* | D(\tau; a) : \tau \neq \upsilon \wedge (\forall \sigma \in \mathcal{A}^\infty | \sigma \geqslant \tau; a : \sigma = \upsilon)) \\ \wedge & \neg D(\epsilon) \end{array} \right. \right)$$

$\Rightarrow$ ⟨Axiom of choice on the first $\forall . \tau; a \geqslant \tau; a, t; a; a \geqslant \tau; a$ and domain weakening on the third $\forall$⟩

$$\left(\exists D \left| \begin{array}{ll} & (\exists \sigma' \preccurlyeq \upsilon | : D(\sigma')) \\ : \wedge & (\forall \tau; a \in \mathcal{A}^* | D(\tau; a) : \tau \neq \upsilon \wedge \tau; a = \upsilon \wedge \tau; a; a = \upsilon) \\ \wedge & \neg D(\epsilon) \end{array} \right. \right)$$

$\Rightarrow$ ⟨Definition of $\preccurlyeq$ and domain Weakening on the second $\exists$. Contradiction and transfer for $\forall$⟩

$$\left(\exists D \left| \begin{array}{ll} & (\exists \sigma' \in \mathcal{A}^* | : D(\sigma')) \\ : \wedge & (\forall \tau; a \in \mathcal{A}^* | : D(\tau; a) \Leftrightarrow \text{false}) \\ \wedge & \neg D(\epsilon) \end{array} \right. \right)$$

$\Leftrightarrow$ Renaming $p \Rightarrow \text{false} \Leftrightarrow \neg p$
$(\exists D | : (\exists \sigma \in \mathcal{A}^* | : D(\sigma)) \wedge (\forall \tau; a \in \mathcal{A}^* | : \neg D(\tau; a)) \wedge \neg D(\epsilon))$
$\Leftrightarrow$ Domain Splitting
$(\exists D | : (\exists \sigma \in \mathcal{A}^* | : D(\sigma)) \wedge (\forall \sigma \in \mathcal{A}^* | : \neg D(\sigma)))$
$\Leftrightarrow$ ⟨ Contradiction and $(\exists x | R : \text{false}) \Leftrightarrow \text{false}$ ⟩
false

Having determined that the set of monitorable properties is indeed increased by relying on static analysis to narrow the set of possible executions, one would naturally wonder if this improvement occurs monotonously, i.e. if every time a sequence υ is removed from a set Σ, a new property is added to the set of $T_{=}^{\Sigma}$-enforceable properties. This would be desirable, as it would imply that any effort made to perform or refine a static analysis of the target program would payoff in the form of an increase in the set of enforceable properties.

Unfortunately, this does not bear out, and there are cases where reducing the size of the set of possible sequences does not result in any advantage. As a counter-example, consider a simple system that can only perform two sequences, each containing only one action, either a or b, or output nothing, thus $\Sigma' = \{\epsilon, a, b\}$. Further, let $\Sigma = \{\epsilon, a\}$. The analysis can be limited to properties that differ only with respect to the sequences present in Σ'. Eight such sequence sets exist, each either allows or disallows each of sequences in Σ'. Because the set of possible properties is finite, it is tractable to examine each and determine that w.r.t. the criteria given in Theorem 5.8, the sets of properties enforceable by each is indeed the same.

In order to increase the set of enforceable properties, at least one of the following three conditions must be met,

Theorem 5.11. (*Constraining* Σ) *Let* $\Sigma \subset \Sigma' \subseteq \mathcal{A}^\infty$, *the set of* $T_{=}^{\Sigma'}$ – enforceable properties $\subset T_{=}^{\Sigma}$ – enforceable properties *iff at least one of the three following conditions are met:*

1. $\exists \upsilon \in \Sigma' \setminus \Sigma : \exists \tau \preccurlyeq \upsilon : \tau \neq \epsilon \wedge \exists \upsilon' \geqslant \tau : \upsilon' \in \Sigma' \wedge \upsilon \neq \upsilon'$
2. $\epsilon \notin \Sigma$
3. $\forall \upsilon \in \mathcal{S}' \setminus \mathcal{S}$: there exists a decidable function $f : \mathcal{S}' \times \mathcal{B}$ s.t. $\forall \tau \in \mathcal{S} : f(\tau_0) =$ **true** if $\tau = \upsilon$ and **false** otherwise.

Proof 4. (*if direction*) In each of cases given above, at least one property is in $T_{=}^{\Sigma}$-enforceable but not in $T_{=}^{\Sigma'}$-enforceable.

1. $\exists \upsilon \in \Sigma' \setminus \Sigma : \exists \tau \preccurlyeq \upsilon : \tau \neq \epsilon \wedge \exists \upsilon' \geqslant \tau : \upsilon' \in \Sigma' \wedge \upsilon \neq \upsilon'$
   There are three cases to consider:
   case $\upsilon \prec \upsilon'$: Let $\widehat{\mathcal{P}}(\sigma) \Leftrightarrow (\sigma = \epsilon \vee \sigma = \upsilon' \sigma)$. This property is not in $T_{=}^{\Sigma'}$-enforceable since the valid sequence υ' has an invalid prefix in Σ'. The property is enforceable over Σ by an automaton which aborts every execution on ϵ, the only valid sequences in Σ.
   case $\upsilon' \prec \upsilon$: Conversely, in this case, the property $\widehat{\mathcal{P}}(\sigma) \Leftrightarrow (\sigma = \epsilon \vee \upsilon)$ is not in $T_{=}^{\Sigma'}$-enforceable, but is $T_{=}^{\Sigma}$-enforceable.
   case $\upsilon' \not\preccurlyeq \upsilon \vee \upsilon \not\preccurlyeq \upsilon'$: Let ν be the smallest sequence in $\Sigma' \setminus \Sigma$ s.t. $\nu \succ \tau \wedge \nu \preccurlyeq \upsilon'$. Let $\widehat{\mathcal{P}}$ be the property $\widehat{\mathcal{P}}(\sigma) \Leftrightarrow (\sigma \not\preccurlyeq \upsilon \vee \sigma = \epsilon)$. This property is not in $T_{=}^{\Sigma'}$-enforceable since the monitor cannot abort on any prefix of ν without losing either transparency or correctness. The property is trivially enforceable over $T_{=}^{\Sigma}$-enforceable.
2. $\epsilon \notin \Sigma$
   There are two cases to consider:
   case $\exists \upsilon' \in \Sigma : \exists \upsilon \in \Sigma' \setminus \Sigma : \upsilon' \prec \upsilon$: As was the case above, the property $\widehat{\mathcal{P}}(\sigma) \Leftrightarrow (\sigma = \epsilon \vee \upsilon)$ is not in $T_{=}^{\Sigma'}$-enforceable, but is $T_{=}^{\Sigma}$-enforceable.
   case $\neg \exists \upsilon' \in \Sigma : \exists \upsilon \in \Sigma' \setminus \Sigma : \upsilon' \prec \upsilon$: Let $\widehat{\mathcal{P}}(\sigma) \Leftrightarrow (\sigma \not\preccurlyeq \upsilon)$. This property is not in $T_{=}^{\Sigma'}$-enforceable since sequence υ has no valid prefix. The property is trivially $T_{=}^{\Sigma}$-enforceable.
3. $\forall \upsilon \in \prime \setminus$: there exists a decidable function $f : \prime \times \mathcal{B}$ s.t. $\forall \tau \in$: $f(\tau_0) =$ **true** if $\tau = \upsilon$ and **false** otherwise.

Let $\widehat{\mathcal{P}}$ be defined such that $\widehat{\mathcal{P}}(\sigma) \Leftrightarrow (\sigma \not\preccurlyeq \upsilon \vee \sigma = \epsilon)$. Let the input sequence be ν. Assume further that none of the two cases mentioned above apply. If the function described above does not exist, then the monitor cannot recognize on the first action of the sequence that the input is invalid. Since ò is the only valid sequence prefix of the input, the monitor has no detector to indicate when the abort the execution, and thus cannot enforce the property. This property is trivially enforceable over.

(only if direction)

We show that the set of $T_{=}^{\Sigma'}$-enforceable properties $= T_{=}^{\Sigma}$-enforceable properties if the following three conditions are met

1. $\forall \upsilon \in \Sigma' \setminus \Sigma : \forall \tau \preccurlyeq \upsilon : \tau \neq \epsilon \Rightarrow \exists \upsilon' \succcurlyeq \tau : \upsilon' \in \Sigma' \wedge \upsilon \neq \upsilon'$
2. $\epsilon \in \Sigma$
3. $\forall \upsilon \in \mathcal{S}' \setminus \mathcal{S}$: there exists a decidable function

$f : \mathcal{S}' \times \mathcal{B}$ s.t. $\forall \tau \in \mathcal{S} : f(\tau_0) =$ **true** if $\tau = \upsilon$ and **false** otherwise.

Let $\widehat{\mathcal{P}}$ be a property in $T_{=}^{\Sigma}$-enforceable. By definition, there exists a decidable predicate D over the sequences of Σ meeting the requirements of Theorem 5.8. For all $\sigma \in \Sigma', a \in \mathcal{A}, \upsilon \in \Sigma' \setminus \Sigma$, the predicate D^* is defined as follows

$$D^*(\tau; a) = \begin{cases} D(\tau; a), & \text{if it is defined;} \\ \text{true}, & \text{if } \neg\widehat{\mathcal{P}}(f(\tau; a)) \text{false} \quad \text{otherwise} \end{cases}$$

The intuition behind the proof is to show that for any property that is $T_{=}^{\Sigma}$-enforceable, a decidable predicate over $\mathcal{A}^*$ meeting the criteria given in Theorem 5.8 can be given, thus rendering the property $T_{=}^{\Sigma'}$-enforceable. First, observe that any property $T_{=}^{\Sigma}$-enforceable is necessarily reasonable since $\epsilon \in \Sigma$. Condition 3 above ensures that a decidable predicate exists which can halt the execution on some prefix of any sequence υ In $\Sigma' \setminus \Sigma$, and since, $\widehat{\mathcal{P}}(\epsilon)$, this prefix is valid. Furthermore, from condition one, ϵ cannot be prefix of any sequence other than υ. Finally, observe that it must be possible to detect from the onset that the current execution is a prefix of υ, since otherwise, a property for which υ has no valid prefix other than ϵ would be $T_{=}^{\Sigma}$-enforceable but not $T_{=}^{\Sigma'}$-enforceable.

$\widehat{\mathcal{P}} \in T_{=}^{\Sigma} -$ enforceable

$\Leftrightarrow \langle$Theorem 5.8$\rangle$

$$\left(\exists D \left| \begin{array}{l} (\forall \sigma \in \Sigma | \neg\widehat{\mathcal{P}}(\sigma) : (\exists \sigma' \preccurlyeq \sigma | : D(\sigma'))) \\ : \wedge \ (\forall \tau; a \in \mathcal{A}^* | D(\tau; a) : \widehat{\mathcal{P}}(\tau) \wedge (\forall \sigma \in \Sigma | \sigma \succcurlyeq \tau; a \wedge \widehat{\mathcal{P}}(\sigma) : \sigma = \tau)) \\ \wedge \ \neg D(\epsilon) \end{array} \right. \right)$$

$\Leftrightarrow \langle$from cond. 3 $\forall \sigma \in \Sigma' \setminus \Sigma : \exists \sigma' \preccurlyeq \sigma : D * (\sigma\prime)\rangle$

$$\left(\exists D \left| \begin{array}{l} (\forall \sigma \in \Sigma' | \neg\widehat{P}(\sigma) : (\exists \sigma' \preccurlyeq \sigma | : D(\sigma'))) \\ : \wedge (\forall \tau; a \in \mathcal{A}^* | D(\tau; a) : \widehat{P}(\tau) \wedge (\forall \sigma \in \Sigma | \sigma \succcurlyeq \tau; a \wedge \widehat{P}(\sigma) : \sigma = \tau)) \\ \wedge \neg D(\epsilon) \end{array} \right. \right)$$

$\Leftrightarrow \langle$since $\epsilon \in \Sigma, \forall \widehat{\mathcal{P}} \in T_{=}^{\Sigma} -$ enforceable $\widehat{\mathcal{P}}(\epsilon)$ and from cond. 1 and the definition of $D^*, D^*(\sigma) \Rightarrow \neq \exists \sigma' \succcurlyeq \sigma : \widehat{\mathcal{P}}(\sigma')\rangle$

$$\left(\exists D \left| \begin{array}{l} (\forall \sigma \in \Sigma' | \neg\widehat{\mathcal{P}}(\sigma) : (\exists \sigma' \preccurlyeq \sigma | : D(\sigma'))) \\ : \wedge \ (\forall \tau; a \in \mathcal{A}^* | D(\tau; a) : \widehat{\mathcal{P}}(\tau) \wedge (\forall \sigma \in \Sigma' \sigma \succcurlyeq \tau; a \wedge \widehat{\mathcal{P}}(\sigma) \sigma = \tau)) \\ \wedge \neg D(\epsilon) \end{array} \right. \right)$$

$\Leftrightarrow \langle$Theorem 5.8$\rangle$

$\widehat{\mathcal{P}} \in T_{=}^{\Sigma'} -$ enforceable

6. Conclusion and future work

The main contribution of this paper is the elaboration of a method aiming at in-lining a security enforcement mechanism in an untrusted program. The security property to be enforced is expressed by a Rabin automaton and the program is modeled by an LTS. The in-lined monitoring mechanism is actually a truncation mechanism allowing valid executions to run normally while halting bad executions before they violate the property.

In this approach, the monitor's enforcement power is extended by giving it access to statically gathered information about the program's possible behavior. This allows us to enforce non-safety properties for some programs. Nevertheless, several cases still exist where the approach fails to find a suitable instrumented code. These are cases where an execution may alternate between satisfying the property or not and could halt in an invalid state, or cases where an invalid execution contains no valid prefixes where the execution could be aborted without also ruling out some valid executions.

Another contribution of this study is to provide a proof that a truncation mechanism that effectively enforces a security property under the equality as an equivalence relation is strictly more powerful in a nonuniform context than in a uniform one.

A more elaborate paradigm dealing with what constitutes a monitor could allow us to ensure the satisfaction of the security property in at least some cases where doing so in currently not feasible. For example, the monitor could suppress a sub-sequence of the program, and keep it under observation until it is satisfied that the program actually satisfies the property and output it all at once. Alternatively, the monitor may be allowed to insert some actions at the end of an invalid sequence in order to guarantee that the sequence is aborted in a valid state. Such monitors are suggested in Bauer et al. (2002), their use would extend this approach to a more powerful framework. Another question that remains open is to determine how often the algorithm will succeed in finding a suitable instrumented code when tested on real programs. We are currently developing an implementation to investigate this question further and hope to gain insights as to which of the above suggested extensions would provide the greatest increase in the set of enforceable properties.

Finally, a distinctive aspect of the method under consideration is that unlike other code instrumentation methods, it induces no added runtime overhead. However, the size of the instrumented program is increased in the order $O(m \times n)$, where m is the size of the original program and n is the size of the property. The instrumentation algorithm itself runs in time $O(p \times c)$, where p is the size of the automaton's acceptance condition and c is the number of cycles in the product automaton. In practice, graphs that abstract programs have a comparatively small number of cycles.

Appendix A. Algorithm

In this section sketches out the algorithm that performs the transformations described in the previous sections. The most interesting function is *Trim*, which eliminates the inadmissible cycles if any and which aborts with **error** if it is not possible to do so.

Algorithm A.1 Synthesizing the Instrumented Program LTS

Input: $\mathcal{R}$ /* *The input Rabin automaton* */
Input: $\mathcal{M}$ /* *The LTS* */
Output: $\mathcal{T}$
1: **let** $\mathcal{R}^P \leftarrow$ AutomataProduct($\mathcal{R}, \mathcal{M}$)
2: **let** $\mathcal{R}^T \leftarrow$ AddHalt($\mathcal{R}^P$) /* *Adding the halt state* */
3: **let** $\mathcal{T} \leftarrow$ Trim($\mathcal{R}^T$) /* *Removing non admissible cycles from* $\mathcal{R}^T$ *with all incident transitions* */

Algorithm A.2 Trim

Input: $\mathcal{R}^T$
Output: $\mathcal{T}$ /* **return** $\mathcal{T}$ *or abort with* **error** */
1: **let** $QT \leftarrow$ BuildScc($\mathcal{R}^T$) /* *Detect the strongly connected components and build the quotient graph QT* */
2: DetectCycles($QT, \mathcal{R}^T$) /* *Detect all the cycles in* $\mathcal{R}^T$ */
3: Annotate($QT, \mathcal{R}^T$) /* *Annotate each scc as A (all cycles admissible), N (all cycles non admissible), B (both), or NC (no cycles)* */
4: SortScc(QT) /* *Sort the scc in reverse topological ordering* */
5: **for all** *scc c in QT* **do**
6: CheckForRemove(c) /*Visit *scc s according to the reverse topological ordering* */
7: **end for**
8: **let** $\mathcal{T} \leftarrow$ update($QT, \mathcal{R}^T$) /* *Build* $\mathcal{T}$ *with states and transitions that have not been removed from QT* */
9: **if** CheckRemovedTransitions($\mathcal{T}, \mathcal{M}$) **then**
10: **return** $\mathcal{T}$
11: **else**
12: **abort** and **return error**
13: **end if**

• CheckRemovedTransitions is a function that scans all the states in $\mathcal{T}$. Let $q = (q_1, q_2) \in \mathcal{T} \cdot Q$ a state in $\mathcal{T}$. Note that $q_1 \in \mathcal{R} \cdot Q$ and $q_2 \in \mathcal{M} \cdot Q$. Let $L(q) = \{a | (\exists q' \in \mathcal{T} \cdot Q | : (q, a, q') \in \mathcal{T} \cdot \delta)\}$ and $L\prime(q) = \{a | (\exists q'_2 \in \mathcal{M} \cdot Q | : (q_2, a, q'_2) \in \mathcal{M} \cdot \delta)\}$.

If there exists at least one state $q \in \mathcal{T} \cdot Q$ such that $L(q) \subset L\prime(q)$ and h is not an immediate successor of q then exit and return **false**.

Algorithm A.3 CheckForRemove(c)

Input: *c*
1: **let** *Annot* ← GetAnnotation(c) /* *Get the scc* annotation */
2: **if** $Annot = A$ **then**
3: **noop** /*Leave *unchanged* */
4: **else if** $Annot = B$ **then**
5: **abort** and **return error**
6: **else if** $Annot = NC$ **then**
7: **if** $Succ(c) = \emptyset$ **then**
8: Remove(c) /* *Remove c with its incident edges. Succ(c) is the set of the successors of c in QT* */
9: **end if**
10: **else if** $Annot = N$ **then**
11: CheckN(c)
12: **end if**

Algorithm A.4 CheckN(c)

Input: *c*
1: **if** $Succ(c) = \emptyset$ **then**
2: Remove(c)
3: **else if** AllAnnotA($Succ(c)$) $= \{H\}$ **then**
4: TryRemoveTransitions(c)
5: RemoveRemainingCycles(c)
6: **else**
7: **abort** and **return error**
8: **end if**

Where

- AllAnnotA(c) returns the set of all successors of *c* annotated *A*,
- TryRemoveTransitions(c) removes the transitions connecting states in *c* that satisfy the following: (q,a,q') is removable if q has h as an immediate successor and if q' is in *c*,
- RemoveRemainingCycles(c) removes the remaining cycles in *c* with all their incident edges. This procedure also removes the states that are no longer accessible with their incident edges.

Appendix B. Proof

In what follows, a sketch of a proof showing that whenever the algorithm succeed in constructing an LTS $\mathcal{T}$ requirements (4.1) and (4.2) hold is given.

Proof of requirement (4.1)

There are two cases to consider, namely the case where $\widehat{\mathcal{P}}(\sigma)$ and the case where $\neg\widehat{\mathcal{P}}(\sigma)$.

- Case 1, $\widehat{\mathcal{P}}(\sigma)$:
  We begin by showing that $\sigma = \tau$. By contradiction, if $\sigma \neq \tau$ then there exists a transition t in $\mathcal{R}^{\mathcal{T}}$, used by σ, but absent in τ. Yet, such a transition could not have been eliminated during the transformation phase of R^T. We will show that this is the case both for transitions that occur inside a *scc* or connecting two different *scc* s. Note that $\widehat{\mathcal{P}}(\sigma)$ means that σ reaches an admissible cycle starting from the initial state.
  - $t = (q1, a, q2)$ with q_1, q_2 in the same *scc* *c*. The *scc* s are treated by the algorithm based on whether the cycles they contain are admissible or not. We examine each possibility in turn.
    * *c* contains only admissible cycles: In such a case, the *scc* is preserved in $\mathcal{T}$ by Algorithm A.3 lines 1–2. Furthermore, since a *scc* can only be removed if it has no admissible successors (Algorithm A.3 lines 7 and 8 and Algorithm A.4 lines 1 and 2), *c* will remain accessible in $\mathcal{T}$.
    * *c* contains both admissible and inadmissible cycles. In such a case, $\mathcal{T}$ cannot be constructed and the algorithm returns **error**.
    * *c* contains only inadmissible cycles. The *scc* is removed only if it has no successor (lines 1 and 2 of Algorithm A.4). In this case, the execution reaching this *scc* cannot be valid. If after crossing *c*, the execution can reach a *scc* with admissible cycles other than *H*, the

algorithm aborts with **error**, (line 7 in Algorithm A.4). Otherwise, the transition t may be removed only if doing so does not remove any initial paths to H. Thus only the transitions that go from immediate predecessors of H to another state in the *scc* are removed (cf. the explanation of the function `TryRemove Transitions`). To sum up, a transition in c is removed only if it cannot allow the execution to reach an admissible cycle.
  - * c contains no cycles. It can only be removed if it has no successors, (lines 6, 7 and 8 in Algorithm A.3). Note that if c has a successor c' with no cycle, it can be showed that from c' we can reach an admissible cycle, otherwise it would have been removed during some previous iteration.
  - $t = (q_1, a, q_2)$ with $q_1 \in c_1$ and $q_2 \in c_2$, $c_1 \neq c_2$. Such a transition is only removed if its destination is a state in a *scc* that is removed.

As discussed above, such an *scc* can never be a part of a valid path.

$\widehat{\mathcal{P}}(\tau)$ follows immediately from $\sigma = \tau$ and $\widehat{\mathcal{P}}(\sigma)$.

- Case 2: $\neg\widehat{\mathcal{P}}(\sigma)$.

  Since the method consists exclusively in removing, rather than adding states and transitions, it is obvious that the execution τ emitted by $\mathcal{T}$ when running σ is either equal to σ, or is a prefix of it. To show that such a sequence τ would always respect the property, two cases must be considered, namely τ is infinite and τ is finite.
  - τ is infinite. In such a case, the proof that $\widehat{\mathcal{P}}(\tau)$ proceeds by contradiction. Let τ be an invalid infinite sequence, τ must enter an inadmissible cycle. Yet, all *scc* containing such cycles were removed from $\mathcal{R}^{\mathcal{T}}$ by algorithm A.4 line 2 or line 5. Where it impossible to remove them, an error message would have been produced without generating $\mathcal{T}$.
  - τ is finite. In this case, σ has been halted in h producing τ or it reached an *end* state, after executing an a_{end} transition. An execution reaching the *end* state satisfy the property, since only executions executions satisfying $\widehat{\mathcal{P}}$ have been kept $\mathcal{R}^{\mathcal{T}}$. Since no transition that could belong to an execution in $\mathcal{L}_{\mathcal{M}}$ have been removed without being sure that the origin of the removed transition is a state that is a predecessor of h, thus we can be sure that we have $\widehat{\mathcal{P}}(\tau)$.

Proof of requirement (4.2). The first half on the conjunction is immediate from the construction process of $\mathcal{T}$. Since no new states or transitions have been added, safe those needed to abort the execution when needed, it follows than any execution that remains in $\mathcal{T}$ was already present in $\mathcal{M}$. Once again the cases of τ being finite or infinite must be examined separately.

- τ is infinite. In such a case, τ can only be invalid if it enters an inadmissible cycle. As discussed above, all such cycles were removed from $\mathcal{T}$ by Algorithm A.4 line 2 or line 5.
- τ is finite. Likewise, it has already been ascertained that any such sequence must be valid, since it necessarily ends in a safe *end* or halt state.

REFERENCES

Aho AV, Sethi R, Ullman JD. Compilers, principles, techniques, and tools. Addison-Wesley; 1986.

Bauer L, Ligatti J, Walker D. More enforceable security policies. In: Proceedings of the foundations of computer security workshop, Copenhagen, Denmark; 2002.

Bauer A, Leucker M, Schallhart C. Monitoring of real-time properties. In: FSTTCS 2006: foundations of software technology and theoretical computer science. Lecture notes in computer science; 2006. p. 260–72.

Beyer D, Henzinger TA, Jhala R, Majumdar R. The software model checker blast: applications to software engineering. International Journal on Software Tools for Technology Transfer (STTT) 2007;9(5–6):505–25.

Bielova N, Massacci F, Micheletti A. Towards practical enforcement theories. In: NordSec; 2009. p. 239–54.

Colcombet T, Fradet P. Enforcing trace properties by program transformation. In: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on principles of programming languages; 2000.

Erlingsson U. The inlined reference monitor approach to security policy enforcement. Ph.D. thesis, Cornell University, Ithaca, NY, USA; 2004.

Erlingsson U, Schneider FB. SASI enforcement of security policies: a retrospective. In: Proceedings of the WNSP: new security paradigms workshop. ACM Press; 2000.

Fong P. Access control by tracking shallow execution history. In: Proceedings of the 2004 IEEE symposium on security and privacy, Oakland, California, USA; 2004.

Hamlen KW, Morrisett G, Schneider FB. Computability classes for enforcement mechanisms. ACM Transactions on Programming Languages and Systems (TOPLAS) 2006;28(1): 175–205.

Khoury R, Tawbi N. Using equivalence relations for corrective enforcement of security policies. In: The fifth international conference mathematical methods, models, and architectures for computer networks security (MMM-ACNS-2010), in press.

Kim M. Information extraction for run-time formal analysis. Ph.D. thesis, University of Pennsylvania; 2001.

Kim M, Viswanathan M, Kannan S, Lee I, Sokolsky O. Java-mac: a run-time assurance approach for java programs. Formal Methods in Systems Design 2004;24(2):129–55.

Langar M, Mejri M. Optimizing enforcement of security policies. In: Proceedings of the foundations of computer security workshop (FCS'05) affiliated with LICS 2005 (Logics in Computer Science); 2005.

Lee, Kannan S, Kim M, Sokolsky O, Viswanathan M. Runtime assurance based on formal specifications. In: Proceedings of the international conference on parallel and distributed processing techniques and applications; 1999.

Ligatti, Reddy S. A theory of runtime enforcement, with results. Tech. Rep. USF-CSE-SS-102809, University of South Florida; Apr 2010.

Ligatti J, Bauer L, Walker D. Edit automata: enforcement mechanisms for run-time security policies. International Journal of Information Security; 2005a.

Ligatti J, Bauer L, Walker D. Enforcing non-safety security policies with program monitors. In: Proceedings of the 10th European symposium on research in computer security (ESORICS), Milan; 2005.

Perrin D, Pin J-E. Infinite words. In: Pure and applied mathematics, vol. 141. Elsevier, ISBN 0-12-532111-2; 2004.

Ramadge PJ, Wonham WM. The control of discrete event systems. IEEE Proceedings: Special issue on Discrete Event Systems 1989;77(1):81–97.

Schneider FB. Enforceable security policies. Information and System Security 2000;3(1):30–50.
Sokolsky O, Kannan S, Kim M, Lee I, Viswanathan M. Steering of real-time systems based on monitoring and checking. In: Proceedings of the fifth international workshop on object-oriented real-time dependable systems, WORDS'99. Washington, DC, USA: IEEE Computer Society; 1999. p. 11.
Talhi C, Tawbi N, Debbabi M. Execution monitoring enforcement under memory-limitations constraints. Information and Computation 2008;206(1):158–84.
Tarjan RE. Depth-first search and linear graph algorithms. SIAM Journal on Computing 1972;1(2):146–60.

Hugues Chabot obtained a Master's degree from Laval University in 2008. He currently works in Computer Security.

Raphaël Khoury is a Ph.D. candidate at Laval University. His research interests center on software safety and dynamic analysis. He is the recipient of a FQRNT scholarship.

Nadia Tawbi obtained her Ph.D. from the Université Pierre et Marie Curie in 1991. She worked at the Bull Research Center, in France from 1996 to 1998. Since 1996, she is professor of Computer Science at Laval University in Québec City, Canada. She has several publications on parallelizing compilers, formal verification and language based security.