Generating In-Line Monitors For Rabin Automata

Hugues Chabot, Raphael Khoury, and Nadia Tawbi

Laval University, Department of Computer Science and Software Engineering,
Pavillon Adrien-Pouliot, 1065, avenue de la Medecine Quebec City, Canada
{hugues.chabot.1,raphael.khoury.1,Nadia.Tawbi}@ulaval.ca

Abstract. A promising solution to the problem of securing potentially malicious mobile code lies in the use of program monitors which are inlined into an untrusted program to produce a new instrumented program that provably respects the security policy captured by that monitor. It is well known that enforcement mechanisms based on Schneider's security automata only enforce safety properties [1]. Yet subsequent studies show that a wider range of properties than those implemented so far could be enforced using monitors. In this paper, we present an approach to producing a model of an instrumented program from a security requirement represented by a Rabin automaton and a model of the program. Based on an a priori knowledge of the program behavior, this approach allows ton enforce in some cases, to enforce more than safety properties. We provide 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.

Key words: Computer Security, Dynamic Analysis,Monitoring Software Safety

1 Introduction

Execution monitoring is an approach to code safety that seeks to allow an untrusted code to run safely by observing its execution and reacting if need be 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 into 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 [2, 3], no further investigations have been pursued in order to take full advantage of this idea in the context of runtime monitoring.

In this paper, we present 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. The policy is stated as a deterministic Rabin automaton, a model which can recognize the same class of languages as non deterministic Büchi automata [4].

In our framework a program execution may be of infinite length representing the executions of programs such as daemons or servers. Finite executions are made infinite by attaching at their end an infinite repetition of a void action.

The use of Rabin automaton is motivated by the need of determinism in order to simplify our method and the associated proofs.

Our approach draws on advances in discrete events system control by [5] and on related subsequent research by Langar and Mejri [6] 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 our approach, the model representing the system's behavior is represented by an LTS and the property to be enforced is stated as a Rabin automaton. The LTS representing the program could be built directly from the control flow graph after a control flow analysis [7, 8].

To sum up, our approach either returns an instrumented program, modeled as a labeled transition system, which provably respects the input security policy or terminates with an error message. While the latter case sometimes occurs, 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. Our approach is thus strictly more expressive.

The rest of this paper is organized as follows. Section 2 presents a review of related work. In Section 3, we define some concepts that are used throughout the paper. The elaborated method is presented in Section 4. In Section 5, we discuss the theoretical underpinnings of our 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 [1], 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 the precise 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, or a truncation 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, Ligatti, Bauer and Walker [3, 9] 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. [10] gave a more precise definition of the set of properties enforceable by the most powerful monitors, while Fong [11] and Talhi et al. [12] expounded on the capabilities of monitors operating under memory constraints. Hamlen et al. [2] , 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 [13], a method is given to enforce both safety and co-safety properties by monitoring.

The first practical application using this framework was developed by Erlingsson and Schneider in [14]. 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 [15], by Langar and Mejri [6] and by Kim et al. [16–19]. 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.

We express the desired security property as a Rabin automaton. 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, we refer to the elements defining an automaton or a model following formalism: the set of states Q of automaton $\mathcal{R}$ is referred to as $\mathcal{R}.Q$ and we leave it as Q when it is clear in the context.

A *path* π, is a finite (respectively infinite) sequence of states $\langle q_1, q_2, ..., q_n \rangle$ (respectively $\langle q_1, q_2, ... \rangle$) such that there exists a finite (respectively infinite) sequence of symbols $a_1, a_2, ..., a_n$ (respectively $a_1, a_2, ...$) called the label of π such that $\delta(q_i, a_i) = q_{i+1}$ for all $i \in \{0, ..., n\}$ (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.

We denote by $set(\pi)$ the set of 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, we write $inf(\pi)$ 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 $inf(\pi)$ is admissible.

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 $\mathcal{L}_{\mathcal{R}}$.

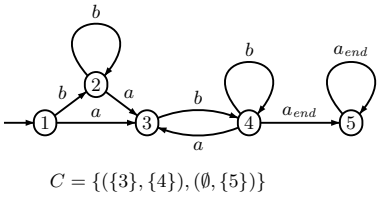
Let $q \in Q$ be a state of $\mathcal{R}$. We say that q is accessible *iff* there exists an initial path (possibly the empty path) that visits q. We say that q is co-accessible *iff* it is the origin of a final path.

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. We write $\tau;\sigma$ for the concatenation of τ and σ. We say that τ is 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 $\pi = \langle q_1, q_2, ..., q_n \rangle$ be a finite path in $\mathcal{R}$. This path is a cycle if $q_1 = q_n$.

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



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

Fig. 1. A Rabin Automaton with acceptance Condition C

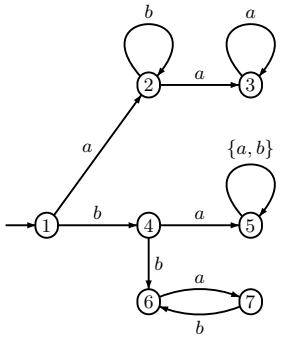


Fig. 2. Example- Labeled transition system

Let us consider Figure 1. It represents a Rabin automaton. In this figure, all the states are 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 $\hat{\mathcal{P}}$ is a predicate on executions. An execution σ is said to be valid or to respect the property if $\hat{\mathcal{P}}(\sigma)$. A Rabin automaton $\mathcal{R}$ represents a security policy $\hat{\mathcal{P}}$ *iff* $\mathcal{L}_{\mathcal{R}} = \{\sigma | \hat{\mathcal{P}}(\sigma)\}$, the set of executions that satisfy the security policy. Abusing the notation, we extend the application of $\hat{\mathcal{P}}$ to a set of sequences, thus if Σ is a set of sequences $\hat{\mathcal{P}}(\Sigma)$ means that all the sequences of Σ satisfy $\hat{\mathcal{P}}$.

4 Method

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

Following [20, 2, 9], we consider that 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. We revisit and expand these ideas in Sections 4.3 and 5.We illustrate each step of our approach using an example program and a security policy.

4.1 Property Encoding

As mentionned earlier, the desired security property is stated as a Rabin automaton. The security property $\hat{\mathcal{P}}$ to which we seek to conform the target program is modeled by the Rabin automaton in Figure 1, over the alphabet $\mathcal{A} \cup \{a_{end}\}$ 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 $\sigma; (a_{end})^{\omega}$. The language accepted by this automaton is the set of executions that containing 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 $\sigma = \tau; (a_{end})^{\omega}$ with $\tau \in \mathcal{A}^*$ is such that $\hat{\mathcal{P}}(\sigma)$ we say that $\hat{\mathcal{P}}(\tau)$.

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). Formally, a 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 $\sigma; (a_{end})^\omega$.

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 we ignore the numbers $i \in \mathbb{N}$ associated with the actions. 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, we can thus consider only deterministic LTSs. Furthermore, we focus exclusively on infinite length executions.

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

As with the Rabin Automata, we define a *path* π as 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 in-lined 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.

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 $\hat{\mathcal{P}}$.
3. Check if $\mathcal{R}^T$ could be used as a truncation automaton and produce a LTS $\mathcal{T}$ modeling the program instrumented by a truncation mechanism otherwise produce $\mathbf{error}$.

The following sections give more details on each step.

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.

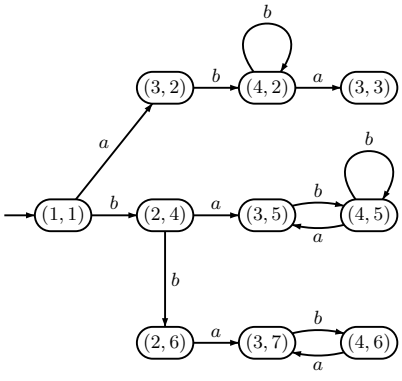
Given a property automaton $\mathcal{R} = (\mathcal{R}.Q, \mathcal{R}.q_0, \mathcal{R}.\delta, \mathcal{R}.C)$ and a Labeled Transition system $\mathcal{M} = (\mathcal{M}.Q, \mathcal{M}.q_0, \mathcal{M}.\delta)$ the automaton $\mathcal{R}^P$ is constructed as follows:

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

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

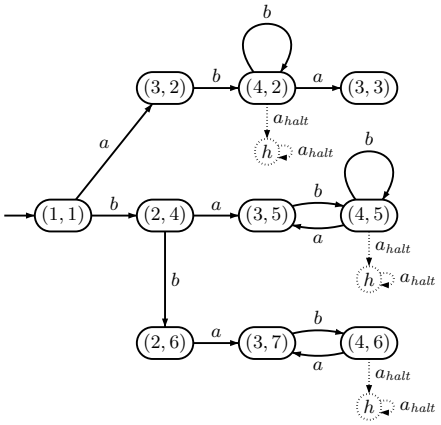
- $\mathcal{R}^P.C = \bigcup_{(L,U)\in\mathcal{R}.C}\{(L \times \mathcal{M}.Q, U \times \mathcal{M}.Q)\}$

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



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

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

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, we extract a deterministic automaton, $\mathcal{T} = (\mathcal{T}.Q, \mathcal{T}.q_0, \mathcal{T}.\delta)$, 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 we seek to construct. 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 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 $\mathcal{M}$, only some of them satisfy the security property. We add a transition, labeled a_{halt}, to a new state h 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 $\mathcal{R}.\delta(q_1, a_{end})$ 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. We have to be careful in choosing h and a_{halt} such that $h \notin \mathcal{R}.Q \cup \mathcal{M}.Q$ and $a_{halt} \notin \mathcal{A}$ the alphabet of actions.

We refer to this updated version of $\mathcal{R}^P$ as $\mathcal{R}^T$, built from $\mathcal{R}^P$ as follows :

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

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

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

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

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 we build an instrumented program 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 $\hat{\mathcal{P}}$ 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 $\mathcal{T}$, we prune $\mathcal{R}^T$ 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, we need to ascertain that $\mathcal{T}$ aborts the execution of every sequence of $\mathcal{L}_{\mathcal{M}}$ not satisfying $\hat{\mathcal{P}}$ and that $\mathcal{T}$ generates only executions satisfying $\hat{\mathcal{P}}$.

We can now restate the correctness requirements of our approach. 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.

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

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

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 [20, 2, 9], 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. We can thus prove that for any invalid sequence present in the original model, the instrumented program outputs a valid prefix of that sequence.

Our 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 $\hat{\mathcal{P}}$. 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 $\hat{\mathcal{P}}$ 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 our approach fails, returning **error**. The underlying idea of the subsequent manipulation is thus to check whether we can trim $\mathcal{R}^T$ by removing bad cycles but without also removing the states and transitions required to ensure transparency.

The following steps show how we perform the trim procedure.

The nest step is to determine the strongly connected components (*scc*) in the graph representing $\mathcal{R}^T$ using Tarjan's algorithm [21]. We then examine each *scc* and mark it as containing either only admissible cycles, only inadmissible cycles, both types of cycles, or no cycles (in the trivial case).To perform this last operation, we have developed heuristics based on the notion that graphs which model programs are structured. A discussion of these heuristics is however beyond the scope of this paper.

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 $\mathcal{R}^T.\delta(q_1, a) = q_2$. 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. We determine for each one whether it should be kept intact, altered or removed.

In the 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 Figure 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 $\hat{\mathcal{P}}$. Eliminating it would prevent the enforcement mechanism from being transparent. In our example in Figure 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, we are generally forced to return **error**. However, in some cases, we can 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 our example, the *scc* containing the states $(3, 7)$ and $(4, 6)$ has only non admissible cycles and H is its only successor. We can eliminate this *scc* and halt with **error** at this point. Yet, if we observe that eliminating the transition $((4, 6), a, (3, 7))$ would break the inadmissible cycle, we can eliminate that transition and keep 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 $\hat{\mathcal{P}}$. We have no means to dynamically check whether the execution is going to leave a cycle or not. Thus, in this case we must abort with **error**. In the example given in Figure 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 we have to abort the algorithm with **error** in such cases.

After removing all the *scc* with inadmissible cycles and provided we have not aborted, we can be sure that an instrumented program built from $\mathcal{T}$ would not contain any infinite length execution which does not respect the security property. We must still verify that whenever the execution is halted, the partial sequence emitted satisfies $\hat{\mathcal{P}}$.

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 $\hat{\mathcal{P}}$. Hence, the verification merely consists in checking whether we have removed transitions from states that are not immediate predecessors of h; if such is the case we have to abort with **error**. More precisely, for a state $q = (q_1, q_2)$ in $\mathcal{T}$ we have to check

whether it is possible from q_2 in $\mathcal{M}$ to perform actions that are not possible from q; if this is the case, q must have h as immediate successor; otherwise, we have no other option than to terminate the algorithm without returning a suitable LTS and with an error message.

We may also remove the transitions of the form (h, a_{halt}, h) and (q, a_{end}, q), where $q \in \mathcal{R}^T.Q$.

4.4 Additional Example

Throughout this section, we have illustrated each step of our approach with an example that was carefully crafted to highlight some of the behavior our algorithm may encounter when in-lining a monitor into a target program. As some aspects of the behavior lead to the rejection of the target program, we were unable to show, using this example, the final result of our approach. 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 Figure 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.
- 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.

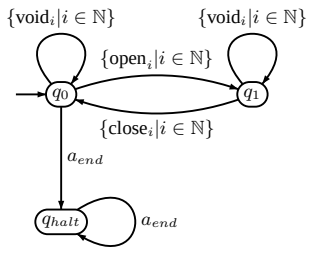


Fig. 5. Example 2, Rabin automaton

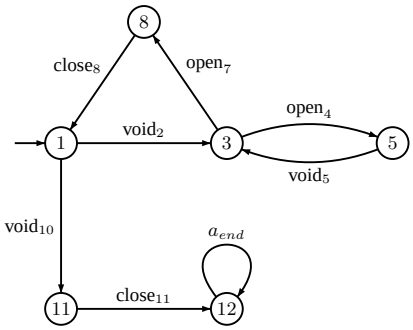


Fig. 6. Example 2, LTS

Figure 6 shows a LTS which approximates the behavior of the program whose execution we wish to monitor. 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 Figure 7.

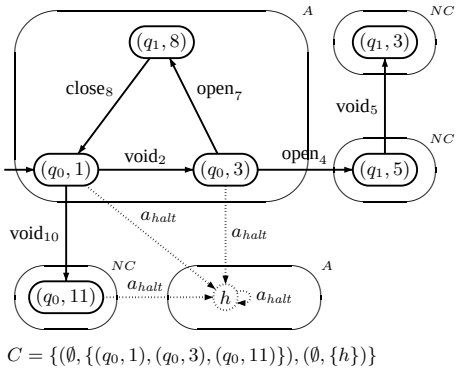


Fig. 7. Example 2, Transformed product automaton

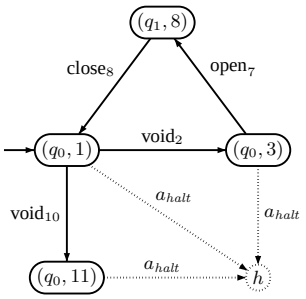


Fig. 8. Example 2, Truncation automaton

In Figure 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.

We omit 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 our approach in this case, namely a LTS that could be transformed into a provably secure program. This LTS is presented in Figure 8.

5 Mechanism's Enforcement power

In this section, we show that non-uniform 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 we discussed in the previous sections, following the notations adopted in [3, 9]. We specify the enforcement mechanism behavior of a security automaton S by judgments of the form

$(q,\sigma)\xrightarrow{\tau}_S(q',\sigma')$ where q is the current state of the automaton; σ is the attempted execution; q' 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 1 (Multi-step semantics).
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 $\tau,\tau'\in\mathcal{A}^*$ we have

$$(q,\sigma)\overset{\varepsilon}{\Longrightarrow}_S(q,\sigma) \tag{5.1}$$

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

We are now able to give the definition of what a security enforcement mechanism is. Intuitively, we can think of security enforcement mechanisms 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 2 (Transformation). *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*

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

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

We have seen that a security enforcement mechanism must respect two properties namely soundness and transparency. The former requires that no invalid execution be permitted, while the latter states that all valid executions must be transformed into semantically equivalent executions. But for enforcement to be meaningful, the notion of equivalence must be constrained. Otherwise, one might argue, for instance, that the empty sequence ϵ is equivalent to every valid execution, and enforce *any* property by aborting every execution at its onset.

Instead, we argue that two executions $\tau,\sigma\in\mathcal{A}^\infty$ are equivalent if there exists a reflexive, symmetric and transitive, equivalence relation $\cong$ s.t. $\tau\cong\sigma$.

We can now state formally what it means for an enforcement mechanism to effectively enforce a security property

Definition 3 (effective$\overset{\Sigma}{\cong}$Application). *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 $\hat{\mathcal{P}}$ 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*

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

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

$$\hat{\mathcal{P}}(\sigma)\Longrightarrow\sigma\cong\tau \tag{5.7}$$

Informally, a security automaton *enforces effectively*$_{\cong}$ 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 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.*

Our approach is built around the idea, first suggested by Ligatti et al. in [3, 9], 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.

We can now state this idea more formally.

Theorem 1. *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 we have*

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

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.The proof has been omitted for space consideration.

Corollary 1. *Let $\mathcal{S}$ be a class of security automaton. For all execution trace set $\Sigma \subseteq \mathcal{A}^{\infty}$ we have*

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

Corollary 1 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 our approach is at least as powerful as those previously suggested in the literature that we built around that last framework.

It would be interesting to prove that for all security automaton classes, $\mathcal{S}$ and for all equivalence relations $\cong$, we have $\mathcal{S}^{\mathcal{A}^{\infty}}_{\cong}$-enforceable $\subset \mathcal{S}^{\Sigma}_{\cong}$-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 our approach, we focus both on a specific class of security automata and on a specific equivalence relation. In our 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 [1]. These are monitors which, when presented with a potentially invalid sequence, have no option but to abort the execution.

Definition 5 (Truncation Automaton). *A truncation automaton is a security automaton where* $\delta : Q \times \mathcal{A} \rightarrow Q \cup \{halt\}$ *and* $halt \notin Q$.

Furthermore, we use syntactic equivalence (=) as the equivalence relation between valid traces.

We can now state the central theorem of this paper, that the enforcement power of the truncation automaton is strictly greater in the nonuniform context than in the uniform context, when we consider =-enforcement.

Theorem 2. *For all set of traces* $\Sigma \subset \mathcal{A}^\infty$ *we have*

$$\mathsf{T}_{=}^{\mathcal{A}^\infty}\text{-enforceable} \subset \mathsf{T}_{=}^{\Sigma}\text{-enforceable} \tag{5.10}$$

The proof is based on the following observations. First, it has been shown in [1, 3] that a property is $\mathsf{T}_{=}^{\mathcal{A}^\infty}$-enforceable iff it is a safety property. Second. Let $\hat{\mathcal{P}}$ be a security property, $\hat{\mathcal{P}}$ is trivially enforceable on Σ *iff* for every sequence $\sigma \in \Sigma$, $\hat{\mathcal{P}}(\sigma)$. The proof thus consists in showing that for any $\Sigma \subset \mathcal{A}^\infty$, a nonsafety property can be stated, and trivially enforced.More specifically, this proof seeks to demonstrate for a sequence $\upsilon \in \mathcal{A}^\infty$ s. t. $\upsilon \notin \Sigma$ the non-safety security property $\hat{\mathcal{P}}(\sigma) \Longleftrightarrow (\sigma \neq \upsilon)$ for all $\sigma \in \mathcal{A}^\infty$ is $\mathsf{T}_{=}^{\Sigma}$-enforceable. The proof has been omitted for space consideration.

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 a 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 our 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 our 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 non uniform 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 [3],

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, ours induces no added runtime overhead. However, the size of the instrumented program is increased in the order $\mathcal{O}(m \times n)$, where m is the size of the original program and n is the size of the property. The instrumentation algorithm itself runs in time $\mathcal{O}(p \times c)$, 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.

References

1. F. B. Schneider, "Enforceable security policies," *Information and System Security*, vol. 3, no. 1, pp. 30–50, 2000.
2. K. W. Hamlen, G. Morrisett, and F. B. Schneider, "Computability classes for enforcement mechanisms," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 28, no. 1, pp. 175–205, 2006.
3. L. Bauer, J. Ligatti, and D. Walker, "More enforceable security policies," in *proceedings of th Foundations of Computer Security Workshop*, Copenhagen, Denmark, Jul. 2002.
4. D. Perrin and J.-E. Pin, *Infinite Words*, ser. Pure and Applied Mathematics. Elsevier, 2004, vol. 141, ISBN 0-12-532111-2.
5. P. J. Ramadge and W. M. Wonham, "The control of discrete event systems," *IEEE Proceedings: Special issue on Discrete Event Systems*, vol. 77, no. 1, pp. 81–97, Jan. 1989.
6. M. Langar and M. Mejri, "Optimizing enforcement of security policies," in *proceedings of the Foundations of Computer Security Workshop (FCS'05) affiliated with LICS 2005 (Logics in Computer Science)*, June-July 2005.
7. A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.
8. D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker BLAST: Applications to software engineering," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 9, no. 5-6, pp. 505–525, 2007.
9. J. Ligatti, L. Bauer, and D. Walker, "Edit automata: Enforcement mechanisms for run-time security policies," *International Journal of Information Security*, 2004.
10. ——, "Enforcing non-safety security policies with program monitors," in *proceedings of the 10th European Symposium on Research in Computer Security (ESORICS)*, Milan, Sep. 2005.
11. P. Fong, "Access control by tracking shallow execution history," in *proceedings of the 2004 IEEE Symposium on Security and Privacy*, Oakland,California, USA, May 2004.
12. C. Talhi, N. Tawbi, and M. Debbabi, "Execution monitoring enforcement under memory-limitations constraints," *Information and Computation*, vol. 206, no. 1, pp. 158–184, 2008.
13. A. Bauer, M. Leucker, and C. Schallhart, "Monitoring of real-time properties," in *FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science*, ser. Lecture Notes in Computer Science, 2006, pp. 260–272.
14. U. Erlingsson and F. B. Schneider, "SASI enforcement of security policies: A retrospective," in *proceedings of the WNSP: New Security Paradigms Workshop*. ACM Press, 2000.

15. T. Colcombet and P. Fradet, “Enforcing trace properties by program transformation,” in *proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Jan. 2000.
16. M. Kim, “Information extraction for run-time formal analysis,” Ph.D. dissertation, University of Pennsylvania, 2001.
17. M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky, “Java-mac: A run-time assurance approach for java programs,” *Formal Methods in Systems Design*, vol. 24, no. 2, pp. 129–155, 2004.
18. I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan, “Runtime assurance based on formal specifications,” in *proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.
19. O. Sokolsky, S. Kannan, M. Kim, I. Lee, and M. Viswanathan, “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.
20. U. Erlingsson, “The inlined reference monitor approach to security policy enforcement,” Ph.D. dissertation, Cornell University, Ithaca, NY, USA, 2004.
21. R. E. Tarjan, “Depth-first search and linear graph algorithms,” *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146–160, 1972.

A Algorithm

In this section, we sketch 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* sccs *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 $q = (q_1, q_2) \in \mathcal{T}.Q$ a state in $\mathcal{T}$. Note that $q_1 \in \mathcal{R}.Q$ and $q_2 \in \mathcal{M}.Q$. Let $L(q) = \{a | (\exists q' \in \mathcal{T}.Q | : (q, a, q') \in \mathcal{T}.\delta\})$ and $L'(q) = \{a | (\exists q'_2 \in \mathcal{M}.Q | : (q_2, a, q'_2) \in \mathcal{M}.\delta)\}$.
If there exists at least one state $q \in \mathcal{T}.Q$ such that $L(q) \subset L'(q)$ and h is not an immediate successor of q then exit and return **false**.

Algorithm A.3 CheckForRemove(c)

Input: c

1: **let** $Annot \leftarrow$ 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.

B Proof

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

Proof of requirement 4.1

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

– Case 1, $\hat{\mathcal{P}}(\sigma)$:
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 $\hat{\mathcal{P}}(\sigma)$ means that σ reaches an admissible cycle starting from the initial state.
  - t = $(q1, a, q2)$ with $q1, q2$ in the same *scc* c. The *scc*s are treated by our 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 (A.3 lines 7 and 8 and A.4 lines 1 and 2), c will remain accessible in $\mathcal{T}$.
    * c contains both admissible and inadmissible cycles. In such a case, we cannot construct $\mathcal{T}$ and are forced to return **error**.
    * c contains only inadmissible cycles. The *scc* is removed only if it has no successor (lines 1 and 2 of 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 we have to abort with **error**, (line 7 in A.4). Otherwise, the transition t may be removed only if doing so does not remove any initial paths to H. We thus remove only the transitions that go from immediate predecessors of H to another state in the *scc* (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 A.3). Note that if c has a successor c' with no cycle, we can show that from c' we can reach an admissible cycle, otherwise it would have been removed during some previous iteration.
  - t = $(q1, a, q2)$ 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.

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

– Case 2 : $\neg\hat{\mathcal{P}}(\sigma)$.
Since our 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, we must consider two cases, namely τ is infinite and τ is finite.
  - τ is infinite. In such a case, the proof that $\hat{\mathcal{P}}(\tau)$ 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. And if we were unable 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. We know that an execution reaching the end state satisfy the property, since we have kept in $\mathcal{R}^{T}$ only executions satisfying $\hat{\mathcal{P}}$. We have been careful to not remove transitions that could belong to an execution in $\mathcal{L}_{\mathcal{M}}$ without being sure that the origin of the removed transition is a state that is a predecessor of h, σ could not reach but an end state or h, thus we can be sure that we have $\hat{\mathcal{P}}(\tau)$.

Proof of requirement 4.2 The first half on the conjunction is immediate from the construction process of $\mathcal{T}$. Since we have have not added any new states or transitions, 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 we have to examine the cases of τ being finite or infinite 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, we have already ascertained that any such sequence must be valid, since it necessarily ends in a safe end or halt state.