Using Edit Automata for Rewriting-Based Security Enforcement

Hakima Ould-Slimane[1], Mohamed Mejri[1], and Kamel Adi[2]

[1] Computer Science Department, Laval University, Quebec, (Qc), Canada
[2] Computer Science Department, University of Quebec in Outaouais, (Qc), Canada

Abstract. Execution monitoring (EM) is a widely adopted class of security mechanisms. EM-enforceable security properties are usually characterized by security automata and their derivatives. However Edit automata (EA) have been recently proposed to specify more powerful EMs. Being able to feign the execution of sensitive program actions, these EMs are supposed to enforce more security properties. However, feigning program actions will usually make the program behaving in discordance with its specification since the effects of feigned actions are not reflected in the program states. In this paper we highlight this problem and show how program rewriting[1] can be a reliable enforcement alternative. The paper contribution is mainly a semantics foundation for program rewriting enforcement of EA-enforceable security properties.

Keywords: Execution monitoring, edit automata, security properties, program rewriting.

1 Introduction

Execution Monitoring (EM) is recognized as an efficient and a powerful class of security mechanisms. An EM enforces a security property by intercepting the property-relevant events and performing an intervention procedure when a program is about to violate the property being enforced [2]. However, the efficiency of EM comes with a significant time and memory overhead. The overhead of monitoring programs is generally attributed to the software and/or hardware artifacts needed for (1) intercepting security relevant actions, (2) analyzing the program execution history (trace), and (3) performing the intervention procedure. However a significant overhead can be avoided if the internal structure of the controlled program is known to the EM. Indeed, statically analyzing the controlled program can reveal that many security-relevant invocations, that the EM is controlling by default during execution, are "innocent" and hence can be safely ignored during execution. In addition, the analysis of the execution history can be simplified by taking into consideration the current program step. All these motivated the recent introduction of the program rewriting approach

[1] Program rewriting studied in this paper refers to the class of security enforcement mechanisms introduced in [1] and has no direct relation with rewriting theory.

E. Gudes, J. Vaidya (Eds.): Data and Applications Security 2009, LNCS 5645, pp. 175–190, 2009.

as alternative to EM [3,1]. In this approach, a security property is enforced on a program by rewriting it to produce a more restrictive one that obeys to the enforced property. In [1], program rewriting is identified as an extremely powerful technique which can be used to enforce properties beyond those enforceable by execution monitors.

A commonly adopted rewriting approach consists in taking an EM, usually specified by a security automaton [2], and embedding it into the program [3,4]. Based on the contributions of [2,5], the security enforcement community shared the belief that any EM-enforceable property is a safety, or equivalently a prefix-closed, property. This belief is based on the fact that an EM can intervene to a potential property violation by only halting the execution of the controlled program. However this belief has been disproved later in [6] by recognizing more powerful EMs having the ability of feigning intercepted actions or inserting (executing) actions on behalf of controlled programs. We call them rewriting-based EMs and they are specified using Edit Automata [7,6]. Feigning actions means preventing their execution in a way that cannot be detected by the controlled program. To enforce a non-safety property, an edit automaton feigns the intercepted actions as long as the entire sequence read before does not obey the enforced property. When it recognizes a safe sequence, it reinserts (executes) all the feigned actions in the same order of their interception.

1.1 Problem of Feigned Actions in Security Enforcement

The efficiency of edit automata in security enforcement is due to their ability to feign potentially dangerous actions and actually execute them later when a safe sequence is recognized. However, not all program actions can be feigned. Indeed, this weakness has been clearly mentioned and admitted in [7,6]. In addition, feigning actions should be performed transparently such that (1) it should not be detected by the controlled program and (2) it should maintain the controlled program in a coherent state. Let ϕ_1 be the property stating that any action *inc* incrementing the value of some variable x should be followed immediately by an action sending the value of that variable. Let assume that the EM enforcing ϕ_1 is intercepting the actions of the program P (Version A) below:

```
1: x := 3; y := 4;                1: x := 3; y := 4;
2: If (x>1) then inc(y);          2: If (x>1) then
3: If (y>4) then send(y);         3:   If (y+1>4) then
                                  4:                {inc(y); send(y);}
                                  5: Else If (y>4) then   send(y);
 Version A                             Version B
```

When intercepting the $inc(y)$ action (step 2), the EM cannot accept it since it does not know the P structure, so it should feign it. However, since $inc(y)$ has not been executed, the $send(y)$ action (step 3) cannot be performed since $y = 4$ which does not satisfy $y > 4$. In this case the EM prevented a valid program sequence from being executed. This problem cannot be addressed without modifying the program structure. A possible solution is to rewrite the program as shown in Version B above. In this new version of P, even if the condition of incrementing y is true, i.e., $x > 1$, incrementing y is delayed until the condition of sending y is satisfied, i.e., $y > 4$. However, since the condition $y > 4$ is assumed to be performed on the value of y after incrementation, we modified this condition so it handles $y + 1$ instead of y. This example highlights the limitations of EM enforcement and shows how program rewriting can enforce properties that cannot be enforced by EM.

1.2 Contributions

In this paper we highlight the problem of feigning actions in EM and show how program rewriting can be a reliable enforcement alternative. We propose a rewriting-based formal technique for enforcing security properties on programs. The enforced properties are those targeted by EA. This technique is based on an injection operator over Extended Finite State Machines (EFSMs). Our technique allows to feign actions in a transparent way; a challenging issue which has not been addressed by previous works. Given an untrusted program and an edit automaton enforcing a property, we produce a new version that satisfies the property. In addition, we proved the soundness and the transparency of our technique. Both programs and the edit automata enforcing security properties are specified by EFSMs.

The rest of the paper is structured as follows. Section 2 discusses the related work. Section 3 presents the formal definitions of EFSMs and Edit Automata. Section 4 provides the formal definition of the injection operator. Section 5 gives the main theorems of our approach. Section 6 illustrates the use of our injection operator through an example. Finally, section 7 presents the conclusion and the future work.

2 Related Work

The characterization of EM security enforcement has been first addressed by Schneider in [2]. One important contribution of [2] is the introduction of security automata as formal specification of EM-enforceable security properties. Another contribution of the same work is the identification of prefix-closed properties as the superset of EM-enforceable security properties. The EMs considered by Schneider control the execution of programs, recognize valid executions, and halt the program execution when a security property violation attempt is identified.

In [7,8,9] Ligatti *et al.* investigated enforcing more security properties using EM. To extend the EM enforcement limits identified in [2], they percept EMs as sequence transformers instead of sequence recognizers as adopted by Schneider. The main contribution of this work is the introduction of edit automata to specify more powerful EMs. Ligatti et al. made an assumption that there exists a security-relevant set of actions for which EA can (1) feign the actions execution, and (2) later execute them on behalf of the controlled programs. As consequence, they extend the theoretical limits of EM enforcement from prefix-closed properties to all properties over finite sequences and renewal properties over infinite sequences [6]. However, practical limitation of EA-based EM is the ability to feign the execution of actions without altering valid executions, an issue that we are addressing by the proposed paper.

Hamlen *et al.* [1] provided a taxonomy of the main security enforcement mechanisms classes. This taxonomy covers static enforcement, execution monitoring, and program rewriting. By connecting the taxonomy to the arithmetic hierarchy of computational complexity theory, the authors succeeded to provide an evaluation of the complexity of each enforcement mechanisms class. The main contribution that is directly related to our work is the identification of inlined EM as alternative to conventional EM. In addition, it showed that program rewriting is an extremely powerful technique which can be used to enforce properties beyond those enforceable by execution monitors [1]. Since the paper was prepared in the same period as [7,6], the authors failed to identify the limitations of EA enforcement compared to program rewriting. Indeed, they (1) recognize EA-enforceable properties as subset of program rewriting properties enforcement (2) but did not add formal constraints on the absence of feigned actions effects on the program execution.

3 Formal Apparatus

In this section we present the main formal definitions needed in this paper. We start by defining extended finite state machines then edit automata.

3.1 Extended Finite State Machine

In order to formally capture the behavior of programs and edit automata, we represent them using Extended Finite State Machines. We adopted this model since the definition of an EFSM is expressive enough to represent edit automata as well as programs with guarded actions that may carry variables.

Definition 1 (Extended Finite State Machine). *Let Σ be a set of possible actions, $\mathcal{V}$ a set of variables, $\mathcal{E}$ a set of arithmetic expressions over $\mathcal{V}$, and $\mathcal{G}$ a set of predicates over Σ^* and $\mathcal{E}$. An Extended Finite State Machine over Σ^*, $\mathcal{V}$, and $\mathcal{G}$ is a 3-tuple (S, i, Δ) where:*

- *S is a finite set of states,*
- *i is a the initial state,*
- *$\Delta \subseteq (S \times (\mathcal{G} \times \Sigma^*) \times S)$ is a transition relation,*

The transition relation Δ consists of 3-tuples of the form $(s_1, (g, \sigma), s_2)$ denoted by $s_1 \xrightarrow{(g,\sigma)} s_2$. This represents a transition from state s_1 to state s_2 guarded by the expression g and executing the sequence of actions σ.

3.2 Edit Automata

Edit automata (EA) are introduced by Ligatti *et al.* [8,6]. In this kind of automata, when given a current state and an input action, the edit automaton transition function specifies a new state to enter and a sequence of actions to execute. The transition function specifies the intervention action to take in order to enforce the property. The intervention action can be accepting the input action or feigning it, inserting a sequence of actions, or halting the execution.

An EA A enforcing a property ϕ can be specified by an EFSM, denoted by E_ϕ. For each transition $s \xrightarrow{g/\sigma} d$ of E_ϕ (1) the guard g specifies the set of actions that can be intercepted by the transition and a condition on their parameters (2) the sequence σ specifies the intervention action that the EA should undertake as response to an intercepted action. Usually, the guard g has the form $\alpha = \mathit{Act} \mid \alpha \neq \mathit{Act}$ where α is the variable referring to the action intercepted by A and Act is any program action. If α should be accepted by A then $\sigma = \alpha$ meaning that α will be actually executed by E_ϕ. If α should be feigned by A then $\sigma = \varepsilon$ meaning that E_ϕ will do nothing as answer to an attempt to execute α. If some sequence of actions σ' should be inserted by A then $\sigma = \sigma'$ meaning that σ' will be executed by E_ϕ instead of α. The three transitions of the EFSM of Figure 1 are examples of each of the aforementioned cases. The EFSM of Figure 1 specifies an EA A_1 enforcing the property ϕ_1 defined by the following regular expression:

$$L_{\phi_1} = \{((\neg inc(x))^*.inc(x).send(x))^*\}$$

The property ϕ_1 states that any action inc incrementing the value of some variable x should be followed immediately by an action sending the value of x.

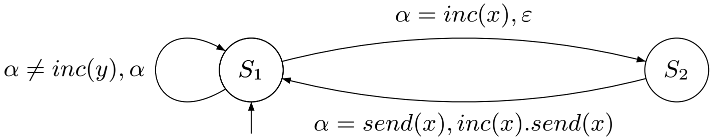

$\alpha = inc(x), \varepsilon$
$\alpha \neq inc(y), \alpha$
S_1
S_2
$\alpha = send(x), inc(x).send(x)$

Fig. 1. An EFSM Specifying the EA Enforcing the Property ϕ_1

4 Injection Operator

In this section, we present our injection operator used to enforce a property ϕ on a program $\mathcal{P}$. Intuitively, this operator inject the EA enforcing ϕ in the program $\mathcal{P}$. The injection operator is a kind of synchronous product over automata. However, since it is used for security enforcement, the order of composition is significant.

Definition 2 (Injection). *Let Σ be a set of possible actions, $\mathcal{V}_\mathcal{P}$ and $\mathcal{V}_\phi$ two disjoint sets of variables, and $\mathcal{G}$ a set of boolean propositions over Σ^* and $\mathcal{V}$. Let $E_\mathcal{P} = (S_\mathcal{P}, i_\mathcal{P}, \Delta_\mathcal{P})$ be an EFSM over Σ^*, $\mathcal{V}_\mathcal{P}$ and $\mathcal{G}$ and $E_\phi = (S_\phi, i_\phi, \Delta_\phi)$ an EFSMs over Σ^*, $\mathcal{V}_\phi$ and $\mathcal{G}$. The EFSM $E_\mathcal{P}$ specifies the behavior of a program*

P while the EFSM E_ϕ specifies the EA enforcing a property ϕ. The ***injection modulo functions*** *$\theta, \mathcal{H}, \mathcal{R}$ denoted by $\circlearrowleft^{\theta}_{\mathcal{H},\mathcal{R}}$ of E_ϕ into $E_\mathcal{P}$ produces the EFSM $E_{\phi 2\mathcal{P}} = (S, i, \Delta)$ over Σ^*, $\mathcal{V}$, and $\mathcal{G}$ where:*

- *$S \subseteq S_\mathcal{P} \times S_\phi \times \Sigma^*$.*
- *$i =< i_\mathcal{P}, i_\phi, \varepsilon >$.*
- *Δ is the set of transitions such that for any state $(s_i, s_k, h) \in S$, if there exists a transition $(s_i, (g_\mathcal{P}, \sigma_\mathcal{P}), s_j) \in \Delta_\mathcal{P}$ and a transition $(s_k, (g_\phi, \sigma_\phi), s_l) \in \Delta_\phi$ then:*

  - $(< s_i, s_k, h >, (g, \sigma), < s_j, s_l, h' >) \in \Delta$ *if* $\sigma_\mathcal{P} \neq \varepsilon$
  - $(< s_i, s_k, h >, (g, \sigma), < s_j, s_k, h' >) \in \Delta$ *if* $\sigma_\mathcal{P} = \varepsilon$

  where :

$$\begin{aligned}
&h' = \mathcal{H}(h, \sigma_\mathcal{P}, \sigma_\phi),\\
&\mathcal{H} : \Sigma^* \times \Sigma^* \times \Sigma^* \longrightarrow \Sigma^*,\\
&\textit{and } (g, \sigma) = \mathcal{R}((g_\mathcal{P}, \sigma_\mathcal{P}), (g_\phi, \sigma_\phi), h),\\
&\mathcal{R} : (\mathcal{G} \times \Sigma^*) \times (\mathcal{G} \times \Sigma^*) \times \Sigma^* \longrightarrow (\mathcal{G} \times \Sigma^*),
\end{aligned}$$

The effects function θ associates to each action $\alpha \in \Sigma$ its effect $\theta(\alpha)$, which is a substitution representing the execution of α. More formally, let $\mathcal{V}(\alpha)$ be the set of variables affected by the execution of some action α. The effect of the execution of α on $\mathcal{V}(\alpha)$ denoted by $\theta(\alpha) = \{\forall x_i \in \mathcal{V}(\alpha), x_i \backslash E_i)\}$ where E_i denotes an expression over $\mathcal{V}$ that will be assigned as new value to the variable x_i after the execution of α. The followings are examples of actions effects:

$$\begin{aligned}
&\theta(inc(x)) = \{x \backslash x + 1\}\\
&\theta(a(z)) = \{z \backslash z^2 + z + 1\}
\end{aligned}$$

The function $\mathcal{H}$ is used to associate with each state of $E_{\phi 2P}$ the sequence of actions h feigned so far as required by E_ϕ. The new sequence h' depends on the sequence h feigned so far and the intervention action required by E_ϕ at this step. The function $\mathcal{R}$ is used to compute the guard and the sequence of actions that should be executed by each transition of $E_{\phi 2P}$. The guard and the actions sequence depend on the sequence σ feigned so far by $E_{\phi 2P}$. More details on the definitions of $\mathcal{H}$ and $\mathcal{R}$ will be provided below:

$$\mathcal{H}(h, \sigma_\mathcal{P}, \sigma_\phi) = \begin{cases} h.\sigma_\mathcal{P} & \text{if } \sigma_\phi = \varepsilon \\ \varepsilon & \text{otherwise} \\ & \end{cases}$$

The history associated with the initial state i is empty because no action has been feigned so far . For the other states, if $\sigma_\phi = \varepsilon$, which means that the current program action ($\sigma_\mathcal{P}$) should be feigned, then the sequence of actions feigned so far (h) should be extended by $\sigma_\mathcal{P}$. If $\sigma_\phi \neq \varepsilon$ then the intervention action needed to enforce ϕ is either accepting the current action or inserting some sequence of actions. In both situations, the history of feigned actions should be reset to ε.

The function $\mathcal{R}$ specifies the actual property enforcement and is defined by:

$$\mathcal{R}((g_{\mathcal{P}},\sigma_{\mathcal{P}}),(g_\phi,\sigma_\phi),h) = \begin{cases} ((g_{\mathcal{P}} \wedge (\sigma_{\mathcal{P}} \Vdash g_\phi))\,\Theta(h), \Omega(\sigma_{\mathcal{P}},\sigma_\phi,h)) & \text{if } \sigma_{\mathcal{P}} \neq \varepsilon \\ ((g_{\mathcal{P}})\,\Theta(h),\varepsilon) & \text{otherwise} \end{cases}$$

where Θ, $\Vdash$, and Ω are defined as follows:

- $\Theta(h)$ is the generalization of function θ on sequences of actions. Indeed, It is the substitution obtained from the history h and applied to the guard $g_{\mathcal{P}}$ and the condition $(\sigma_{\mathcal{P}} \Vdash g_\phi)$ to generate a more restrictive guard. This new guard carries the impact (the effect) of feigning the actions of h on the program variables. Consequently, we call $\Theta(h)$ "the feigning substitution of h". The use of the substitutions $\Theta(h)$ represents the way we are following to feign actions execution. Following this way, if the actions of h are feigned at some execution point in $E_{\phi 2P}$ then, from the property enforcement (E_ϕ) viewpoint, (E_P) will behave according to the EA (E_ϕ) recommendations. However from E_P viewpoint, $E_{\phi 2P}$ will be in an incoherent state since it will be in a state were h is supposed to be already executed, which is not the case. To avoid this incoherence, the substitution $\Theta(h)$ is used to simulate the effects of executing h on the program variables. As a result, the actions in $E_{\phi 2P}$ are executed according to the recommendations of E_ϕ while the program guards controlling the execution of actions reflect the program state as if the actions are executed exactly as recommended by E_P.
  Let $\Theta(h)$ be the substitution to be applied to some guard g, we have:
  - If $h = \varepsilon$ (the history is empty), then we have: $(g)\,\Theta(\varepsilon) = g$.
  - If $h = \alpha_1.\alpha_2 \ldots \alpha_n$ then: $(g)\,\Theta(\alpha_1.\alpha_2 \ldots \alpha_n) = (g)(\theta(\alpha_1) \circ \ldots \circ \theta(\alpha_n)) = (\ldots(((g)\,\theta(\alpha_n))\theta(\alpha_{n-1}))\ldots\theta(\alpha_1))$. For example:
    $$(x > y)\,\Theta(inc(x).dec(y)) = (x > y)\{x\backslash x{+}1\}\circ\{y\backslash y{-}1\} = (x{+}1 > y{-}1)$$
    where $inc(x)$ is the action that increments the value of x, while $dec(y)$ is the action that decrements the value of y.

- "$\Vdash$" is called "the Satisfaction Function", it takes a program sequence (consisting of only one action) and a property guard (which concerns some program action) and returns a guard over program variables. The relation "$\Vdash$" is defined in Table 1 where:
  - $Act(g)$ denotes the function extracting the action from a property predicate g. For example: $Act(\alpha = read(z)) = read(z)$.
  - $mgu(a, Act(g))$ denotes the most general unifier between a and $Act(g)$.
  - $\mathcal{V}(a)$ denotes the set of variables used and/or updated by the action a.
  - $\mathcal{V}(g)$ denotes the set of variables involved in the predicate g.

Table 1. The Satisfaction Function

$$a \Vdash g = \begin{cases} tt & \text{if } g = tt \\ \neg(a \Vdash g') & \text{if } g = \neg g' \\ \mathit{ff} & \text{if } mgu(a, Act(g)) \text{ doesn't exist} \\ Tdyn_{\mathcal{V}(a)}^{\mathcal{V}(g)}(\delta) \wedge (Tpar(g))\delta & \text{if } \exists\delta = mgu(a, Act(g)) \end{cases}$$

Table 2. The Dynamic Test

$$\begin{array}{lll} Tdyn_{\mathcal{V}(a)}^{\mathcal{V}(g)}(\emptyset) & = tt & \\ Tdyn_{\mathcal{V}(a)}^{\mathcal{V}(g)}(\{x \mapsto t\} \cup \delta) & = (x = t) \wedge Tdyn_{\mathcal{V}(a)}^{\mathcal{V}(g)}(\delta) & \text{if } x \in \mathcal{V}(a) \wedge t \notin \mathcal{V}(g) \\ Tdyn_{\mathcal{V}(a)}^{\mathcal{V}(g)}(\{x \mapsto y\} \cup \delta) & = Tdyn_{\mathcal{V}(a)}^{\mathcal{V}(g)}(\delta) & \text{if } x \in \mathcal{V}(a) \wedge y \in \mathcal{V}(g) \\ Tdyn_{\mathcal{V}(a)}^{\mathcal{V}(g)}(\{x \mapsto t\} \cup \delta) & = Tdyn_{\mathcal{V}(a)}^{\mathcal{V}(g)}(\delta) & \text{if } x \in \mathcal{V}(g) \end{array}$$

- $Tdyn_{\mathcal{V}(a)}^{\mathcal{V}(g)}(\delta)$ is a boolean condition on $\mathcal{V}(a)$ and it is extracted from the substitution δ. This substitution is obtained from the unification of a program action and the corresponding action targeted by the property guard. This condition represents the dynamic test that should be checked during execution. $Tdyn_{\mathcal{V}(a)}^{\mathcal{V}(g)}(\delta)$ is defined in Table 2. For example:

  $Tdyn_{\{x\}}^{\{y,z\}}([x \mapsto 5, y \mapsto 3, x \mapsto z]) = (x = 5)$.
- $Tpar(g)$ denotes the function extracting the guard handling the property parameters from the predicate g. For example:
  $Tpar(\alpha = read(z) \wedge z > 2) = (z > 2)$. Hence, the condition $(Tpar(g))\delta$ will reflect the test required by the property on the program variables, by substituting the property parameters with the corresponding program variable, as stated by the substitution δ.

The followings are some examples of computing the satisfaction function:

- $a(x) \Vdash (\alpha = a(z)) = \underline{tt}$. This means that the program action $a(x)$ satisfies the property predicate $(\alpha = a(z))$ with no condition.
- $a(x) \Vdash (\alpha = a(3)) = \underline{(x = 3)}$. This means that the program action $a(x)$ satisfies the property predicate $(\alpha = a(3))$ under the condition $(x = 3)$, which is a dynamic test.
- $a(x) \Vdash (\alpha = b(z)) = \underline{ff}$. This means that the program action $a(x)$ does not satisfy the property predicate $(\alpha = b(z))$.

– Ω is the function that generates the new sequence of actions to be executed at each transition of $E_{\phi 2P}$. It is called "the editing function" and defined by:

$$\Omega(\sigma_{\mathcal{P}}, \sigma_{\phi}, h) \quad = \quad \begin{cases} \varepsilon & \text{if } \sigma_{\phi} = \varepsilon \\ \sigma_{\mathcal{P}} & \text{if } \sigma_{\phi} = \text{“}\alpha\text{”} \\ h.\sigma_{\mathcal{P}} & \text{otherwise} \end{cases}$$

The editing function Ω will take one of the three following decisions:

- Feigning the current program action, by executing no action ($\sigma_{\phi} = \varepsilon$)
- Executing the same action as required by the program ($\sigma_{\phi} =$ “α”)
- Executing a sequence of actions when a valid sequence prefix ($h.\sigma_{\mathcal{P}}$) is reached.

The followings are some examples of the use of Ω:

- $\Omega(a(x), \underline{\varepsilon}, b(y).c(z)) = \varepsilon$ (feigning case).
- $\Omega(a(x), \underline{\alpha}, \varepsilon) = a(x)$ (acceptation case).
- $\Omega(b(3), \underline{a(x).b(y)}, a(z)) = a(z).b(3)$ (insertion case).

Optimization. After the construction of the EFSM $E_{\phi 2P}$, some optimizations will be performed on it in order to eliminate unreachable states. This is done by eliminating all the useless transitions. A transition is considered as useless if it is labeled with (g, σ) where g is equivalent to false (*ff*). If after eliminating useless transitions some nodes become not reachable from the initial state then they will be removed. We remove also each path $(g_m, \sigma_m)...(g_n, \sigma_n)$ in which all the actions have been feigned during the enforcement ($\sigma_m = ... = \sigma_n = \varepsilon$), since the new program will execute no action following those paths.

5 Main Results

The following definitions are needed to prove the theorems presented in this section. Any mention to a path refers to a path starting from the initial state.

Definition 3 (Property Satisfaction). *Let $E_{\mathcal{P}}$ be the EFSM specifying a program $\mathcal{P}$ and E_{ϕ} an EFSM specifying an EA enforcing a property ϕ. We say that $\mathcal{P}$ satisfies Φ, denoted by $\mathcal{P} \models \phi$, if every path in E_P can be extended to a path in E_P that satisfies Φ, let $\tau_{\mathcal{P}} = (g_1, \sigma_1).(g_2, \sigma_2), \ldots (g_n, \sigma_n)$ be this path. Since each σ_i is a sequence of actions, we can write their concatenation as: $\sigma_1.\sigma_2 \ldots \sigma_n = a_1.a_2 \ldots a_m$ where each a_i is an atomic action and $m = |\sigma_1.\sigma_2 \ldots \sigma_n|$.*

The path $\tau_{\mathcal{P}}$ satisfies Φ, if there exists a path $\tau_{\phi} = (g'_1, \sigma'_1).(g'_2, \sigma'_2) \ldots (g'_m, \sigma'_m)$ belonging to E_{Φ} such that:

(A) $mgu(a_1 \ldots a_m, \sigma'_1 \ldots \sigma'_m) \neq \emptyset$.

(B) $g_1 \sqsubseteq (a_1 \Vdash g'_1)$ and $\forall 1 \leq j \leq m. \exists 1 \leq i \leq n$ such that $a_1. \ldots .a_j$ is a prefix of $\sigma_1. \ldots .\sigma_i$ and $(g_1 \bigwedge_{2 \leq l \leq i} (g_l)\Theta(\sigma_1 \ldots \sigma_{l-1})) \sqsubseteq (a_j \Vdash g'_j)\Theta(a_1 \ldots a_{j-1})$

where $\sqsubseteq$ is an ordering relation over boolean expressions. For any two conditions b and b', we say that $b \sqsubseteq b'$, meaning “b is more restrictive than b'”, if there exists a condition b'', such that $b = b' \wedge b''$.

Intuitively, condition (A) states that the execution produced by the program path matches a prefix of a property path regardless of the property conditions. Condition (B) states that the conjunction of all the conditions (guards) controlling the execution of any action of the program at some prefix of the path is more restrictive than the conjunction of those conditions (guards) expected by the property. Since the number of the conditions of the program path can be different from the number of conditions of the property path, the effects of the actions executed before some program/property condition are simulated on them.

Definition 4 (Precise Program Restriction $\precsim$). *Let $E_{\mathcal{P}}$ and $E_{\mathcal{P}'}$ be two EFSMs specifying two programs $\mathcal{P}$ and $\mathcal{P}'$ respectively. We say that $\mathcal{P}$ is a precisely more restrictive version of $\mathcal{P}'$, denoted by $\mathcal{P} \precsim \mathcal{P}'$, if each path belonging to $E_{\mathcal{P}}$ can be extended to some path $\tau_{\mathcal{P}} = (g_1, \sigma_1)\ldots(g_n, \sigma_n)$ belonging to $E_{\mathcal{P}}$ for which there exists a path $\tau_{\mathcal{P}'} = (g'_1, \sigma'_1)\ldots(g'_m, \sigma'_m)$ belonging to $E_{\mathcal{P}'}$ such that:*

(C) $(g_1 \wedge \bigwedge_{2\leq i\leq n}(g_i)\Theta(\sigma_1\ldots\sigma_{i-1})) \sqsubseteq (g'_1 \wedge \bigwedge_{2\leq i\leq m}(g'_i)\Theta(\sigma'_1\ldots\sigma'_{i-1}))$[2]

(D) $\sigma_1\ldots\sigma_n = \sigma'_1\ldots\sigma'_m$.

Intuitively, condition (C) states that the conjunction of all the conditions (guards) controlling the execution produced by the path of program $\mathcal{P}$ is more restrictive than the conjunction of those conditions controlling the same execution produced by the path of program $\mathcal{P}'$. Condition (D) states that any execution of $\mathcal{P}$ can be performed by $\mathcal{P}'$. The definition of "precise program restriction" is needed by Theorem 1 to compare a program with the program resulting from injecting a property into it. Here the resulting program should be more restrictive than the original one in terms of the possible executions and the conditions controlling them. Definition 4 is also needed by Theorem 2 to verify that a any subprogram of $\mathcal{P}$ that satisfies a property is preserved by the property injection.

Definition 5 (Conservative Program Restriction $\preceq$). *Let $E_{\mathcal{P}}$ and $E_{\mathcal{P}'}$ be two EFSMs specifying two programs $\mathcal{P}$ and $\mathcal{P}'$ respectively. We say that $\mathcal{P}$ is a conservatively more restrictive version of $\mathcal{P}'$, denoted by $\mathcal{P} \preceq \mathcal{P}'$, if for each path $\tau_{\mathcal{P}} = (g_1, \sigma_1)\ldots(g_n, \sigma_n)$ belonging to $E_{\mathcal{P}}$ there exists some path $\tau_{\mathcal{P}'} = (g'_1, \sigma'_1)\ldots(g'_n, \sigma'_n)$ belonging to $E_{\mathcal{P}'}$ such that:*

(E) $g_1 \sqsubseteq g'_1 \wedge \forall 2 \leq i \leq n.(g_i)\,\Theta(\sigma_1\ldots\sigma_{i-1}) \sqsubseteq (g'_i)\,\Theta(\sigma'_1\ldots\sigma'_{i-1})$.

(F) $\forall i, 1 \leq i \leq n. \quad \sigma_i = \sigma'_i$.

Intuitively, conditions (E) and (F) together state that for any path of $\mathcal{P}$ there exists some path in $\mathcal{P}'$ (1) executing the same sequence of actions, (2) having the same number of guards that each of which controls the same sequence of

[2] Recall that Θ is the substitution simulating the effect of a sequence of actions on the transitions predicates. See Section 4 for the formal definition of Θ.

action, and (3) each guard of $\mathcal{P}$ is more restrictive than the corresponding guard of $\mathcal{P}'$. The definition of "conservative program restriction" is needed by Theorem 2 to identify inside a program $\mathcal{P}$ all its subprograms that satisfy a property. The theorem will select these subprograms as conservative restrictive versions of $\mathcal{P}$.

The following theorem states that our approach of injecting a property ϕ into a program $\mathcal{P}$ is sound. Soundness means that all the possible executions of the EFSM resulting from injecting ϕ into $\mathcal{P}$ are possible executions of the original EFSM specifying $\mathcal{P}$. It states also that all the possible executions of the resulting EFSM satisfies the property ϕ.

Theorem 1 (Soundness). *Let $\mathcal{P}$ be a program specified by the EFSM $E_{\mathcal{P}}$, Φ a property enforced by the EA specified by the EFSM E_{ϕ}, and $P \circlearrowleft \Phi$ be the program specified by the EFSM $E_{\phi 2\mathcal{P}} = E_{\mathcal{P}} \circlearrowleft^{\theta}_{\mathcal{H},\mathcal{R}} E_{\phi}$. We have:*

(I) $P \circlearrowleft \Phi \precsim P$

(II) $P \circlearrowleft \Phi \models \Phi$

Proof. [3]
Let $\tau = (g_1, \sigma_1).(g_2, \sigma_2) \dots (g_n, \sigma_n)$ be any path of $E_{\phi 2\mathcal{P}}$. It follows, from the definition of a path, that there exist states $(s_0, h_0), (s_1, h_1) \dots (s_{n-1}, h_{n-1})$, (s_n, h_n) such that $(s_0, h_0) \xrightarrow{(g_1,\sigma_1)} (s_1, h_1) \dots (s_{n-1}, h_{n-1}) \xrightarrow{(g_n,\sigma_n)} (s_n, h_n)$ is a legitimate sequence of transitions, starting from (s_0, h_0). From the definition of $\circlearrowleft^{\theta}_{\mathcal{H},\mathcal{R}}$, it follows that there exist states $s^{\mathcal{P}}_0, s^{\mathcal{P}}_1, \dots s^{\mathcal{P}}_n$ and $s^{\phi}_0, s^{\phi}_1, \dots s^{\phi}_n$ and propositions $g^{\mathcal{P}}_1, g^{\mathcal{P}}_2, \dots g^{\mathcal{P}}_n$ and $g^{\phi}_1, g^{\phi}_2, \dots g^{\phi}_n$ such that:

a) there exists a path $\tau_{\mathcal{P}} = (g^{\mathcal{P}}_1, \sigma^{\mathcal{P}}_1).(g^{\mathcal{P}}_2, \sigma^{\mathcal{P}}_2) \dots (g^{\mathcal{P}}_n, \sigma^{\mathcal{P}}_n)$ generated by the following legitimate sequence of transitions, starting from the initial state $s^{\mathcal{P}}_0$ of $E_{\mathcal{P}}$: $s^{\mathcal{P}}_0 \xrightarrow{(g^{\mathcal{P}}_1,\sigma^{\mathcal{P}}_1)} s^{\mathcal{P}}_1 \xrightarrow{(g^{\mathcal{P}}_2,\sigma^{\mathcal{P}}_2)} s^{\mathcal{P}}_2 \dots \xrightarrow{(g^{\mathcal{P}}_n,\sigma^{\mathcal{P}}_n)} s^{\mathcal{P}}_n$.

b) there exists a path $\tau_{\phi} = (g^{\phi}_1, \sigma^{\phi}_1).(g^{\phi}_2, \sigma^{\phi}_2) \dots (g^{\phi}_n, \sigma^{\phi}_n)$ generated by the following legitimate sequence of transitions, starting from the initial state s^{ϕ}_0 of E_{ϕ}: $s^{\phi}_0 \xrightarrow{(g^{\phi}_1,\sigma^{\phi}_1)} s^{\phi}_1 \xrightarrow{(g^{\phi}_2,\sigma^{\phi}_2)} s^{\phi}_2 \dots \xrightarrow{(g^{\phi}_n,\sigma^{\phi}_n)} s^{\phi}_n$

c) from the definition of $\circlearrowleft^{\theta}_{\mathcal{H},\mathcal{R}}$ we have:
   - $\forall i \geq 1$: $g_i = (g^{\mathcal{P}}_i \wedge (\sigma^{\mathcal{P}}_i \Vdash g^{\phi}_i))\Theta(h_{i-1})$
   - $\forall i \geq 1$: $\sigma_i = \Omega(\sigma^{\mathcal{P}}_i, \sigma^{\phi}_i, h_{i-1})$

Proof of (I): To prove (I): $P \circlearrowleft \Phi \precsim P$ we will prove that for each path τ of $E_{\phi 2\mathcal{P}}$, there exists an extension for which there exists a path $\tau_{\mathcal{P}}$ of $E_{\mathcal{P}}$ such that the conditions (C) and (D) of Definition 4 should be satisfied. Let τ be the extension of τ that will be considered by our proof[4]. From (a), we know that for each path τ there exists some path $\tau_{\mathcal{P}}$ of $E_{\mathcal{P}}$. Accordingly the conditions (C) and (D) to be proved are respectively the following:

[3] For space limitations not all the intermediate steps of the proof have been presented.

[4] Recall that any path can be considered as extension of itself.

- $(g_1 \wedge \bigwedge_{2\leq i\leq n}(g_i)\Theta(\sigma_1 \ldots \sigma_{i-1})) \sqsubseteq (g_1^{\mathcal{P}} \wedge \bigwedge_{2\leq i\leq m}(g_i^{\mathcal{P}})\Theta(\sigma_1^{\mathcal{P}} \ldots \sigma_{i-1}^{\mathcal{P}}))$
- $\sigma_1.\ldots.\sigma_n = \sigma_1^{\mathcal{P}}.\ldots.\sigma_n^{\mathcal{P}}$

By replacing the terms by there definitions in c) the two conditions (C) and (D) will be the followings:

(C) $(g_1^{\mathcal{P}} \wedge (\sigma_1^{\mathcal{P}} \Vdash g_1^{\phi})) \wedge \bigwedge_{2\leq i\leq n}(g_i^{\mathcal{P}} \wedge (\sigma_i^{\mathcal{P}} \Vdash g_i^{\phi}))\,\Theta(h_{i-1})$
$\Theta(\Omega(\sigma_1^{\mathcal{P}},\sigma_1^{\phi},h_0)\ldots\Omega(\sigma_{i-1}^{\mathcal{P}},\sigma_{i-1}^{\phi},h_{i-2})) \sqsubseteq g_1^{\mathcal{P}} \wedge \bigwedge_{2\leq i\leq n}(g_i^{\mathcal{P}})\,\Theta(\sigma_1^{\mathcal{P}} \ldots \sigma_{i-1}^{\mathcal{P}})$.
(D) $\sigma_1.\ldots.\sigma_n = \sigma_1^{\mathcal{P}}.\ldots.\sigma_n^{\mathcal{P}}$

d) From the definition of $\mathcal{H}$, we have:
$g_1 = (g_1^{\mathcal{P}} \wedge (\sigma_1^{\mathcal{P}} \Vdash g_1^{\phi}))\,\Theta(h_0) = (g_1^{\mathcal{P}} \wedge (\sigma_1^{\mathcal{P}} \Vdash g_1^{\phi}))\,\Theta(\varepsilon) = g_1^{\mathcal{P}} \wedge (\sigma_1^{\mathcal{P}} \Vdash g_1^{\phi}) \sqsubseteq g_1^{\mathcal{P}}$

Let E_i denote the individual terms at the left side of condition (C):
$(g_i^{\mathcal{P}} \wedge (\sigma_i^{\mathcal{P}} \Vdash g_i^{\phi}))\,\Theta(h_{i-1})\,\Theta(\Omega(\sigma_1^{\mathcal{P}},\sigma_1^{\phi},h_0)\ldots\Omega(\sigma_{i-1}^{\mathcal{P}},\sigma_{i-1}^{\phi},h_{i-2}))$.

According to the value of the history h_{i-1}, we distinguish two cases:

- *Case 1:* $h_{i-1} = \varepsilon$ (i.e., no action has been feigned so far), we have: $\Omega(\sigma_1^{\mathcal{P}},\sigma_1^{\phi},h_0))\ldots\Omega(\sigma_{i-1}^{\mathcal{P}},\sigma_{i-1}^{\phi},h_{i-2}) = \sigma_1^{\mathcal{P}} \ldots \sigma_{i-1}^{\mathcal{P}}$. Then we deduce that: $E_i = (g_i^{\mathcal{P}})\,\Theta(\sigma_1^{\mathcal{P}} \ldots \sigma_{i-1}^{\mathcal{P}}) \wedge (\sigma_i^{\mathcal{P}} \Vdash g_i^{\phi})\,\Theta(\sigma_1^{\mathcal{P}} \ldots \sigma_{i-1}^{\mathcal{P}}) \sqsubseteq (g_i^{\mathcal{P}})\,\Theta(\sigma_1^{\mathcal{P}} \ldots \sigma_{i-1}^{\mathcal{P}})$.
- *Case 2:* $h_{i-1} \neq \varepsilon$, i.e., a sequence of actions has been feigned without being reinserted. This sequence is $h_{i-1} = \sigma_m^{\mathcal{P}} \ldots \sigma_{i-1}^{\mathcal{P}}$ where $1 \leq m \leq i-1$. Consequently, we have: $\Omega(\sigma_1^{\mathcal{P}},\sigma_1^{\phi},h_0)\ldots\Omega(\sigma_{i-1}^{\mathcal{P}},\sigma_{i-1}^{\phi},h_{i-2}) = \sigma_1^{\mathcal{P}} \ldots \sigma_{m-1}^{\mathcal{P}}$. Then, we deduce that the expression E_i becomes:

  $(g_i^{\mathcal{P}})\,\Theta(\sigma_m^{\mathcal{P}} \ldots \sigma_{i-1}^{\mathcal{P}})\,\Theta(\sigma_1^{\mathcal{P}} \ldots \sigma_{m-1}^{\mathcal{P}}) \wedge (\sigma_i^{\mathcal{P}} \Vdash g_i^{\phi})\,\Theta(\sigma_m^{\mathcal{P}} \ldots \sigma_{i-1}^{\mathcal{P}})\,\Theta(\sigma_1^{\mathcal{P}} \ldots \sigma_{m-1}^{\mathcal{P}}) = (g_i^{\mathcal{P}})\,\Theta(\sigma_1^{\mathcal{P}} \ldots \sigma_{i-1}^{\mathcal{P}}) \wedge (\sigma_i^{\mathcal{P}} \Vdash g_i^{\phi})\,\Theta(\sigma_1^{\mathcal{P}} \ldots \sigma_{i-1}^{\mathcal{P}})$
  $\sqsubseteq (g_i^{\mathcal{P}})\,\Theta(\sigma_1^{\mathcal{P}} \ldots \sigma_{i-1}^{\mathcal{P}})$.

From *Case 1* and *Case 2*, we obtain:
$\forall 2 \leq i \leq n.\, \bigwedge_{2\leq i\leq n} E_i \sqsubseteq \bigwedge_{2\leq i\leq n}(g_i^{\mathcal{P}})\,\Theta(\sigma_1^{\mathcal{P}} \ldots \sigma_{i-1}^{\mathcal{P}})$.

By adding the term that we got from d) we obtain:
$(g_1^{\mathcal{P}} \wedge (\sigma_1^{\mathcal{P}} \Vdash g_1^{\phi})) \wedge \bigwedge_{2\leq i\leq n}(g_i^{\mathcal{P}} \wedge (\sigma_i^{\mathcal{P}} \Vdash g_i^{\phi}))\,\Theta(h_{i-1})\,\Theta(\Omega(\sigma_1^{\mathcal{P}},\sigma_1^{\phi},h_0)\ldots$
$\Omega(\sigma_{i-1}^{\mathcal{P}},\sigma_{i-1}^{\phi},h_{i-2})) \sqsubseteq g_1^{\mathcal{P}} \wedge \bigwedge_{2\leq i\leq n}(g_i^{\mathcal{P}})\,\Theta(\sigma_1^{\mathcal{P}} \ldots \sigma_{i-1}^{\mathcal{P}})$ which satisfies condition (C).

Since $h_n \neq \varepsilon$, we have: $\Omega(\sigma_1^{\mathcal{P}},\sigma_1^{\phi},h_0)\ldots\Omega(\sigma_n^{\mathcal{P}},\sigma_n^{\phi},h_{n-1}) = \sigma_1^{\mathcal{P}} \ldots \sigma_n^{\mathcal{P}}$ which satisfies condition (D).

By proving (C) and (D) we have proved that (I): $P \circlearrowleft \Phi \precsim P$.

Proof of (II): To prove (II): $P \circlearrowleft \Phi \models \Phi$ we will prove that for each path $\tau = (g_1,\sigma_1).(g_2,\sigma_2)\ldots(g_n,\sigma_n)$ of $E_{\phi 2\mathcal{P}}$, there exists an extension for which there exists a path τ_ϕ of E_ϕ such that the conditions (A) and (B) of Definition 3 should be satisfied. Let τ be the extension of τ that will be considered by our proof[5]. From a), we know that for each path τ there exists some path $\tau_{\mathcal{P}} = (g_1^{\mathcal{P}}, \sigma_1^{\mathcal{P}}).(g_2^{\mathcal{P}}, \sigma_2^{\mathcal{P}}) \ldots (g_n^{\mathcal{P}}, \sigma_n^{\mathcal{P}})$. such that $\sigma_1 . \ldots . sigma_n = \sigma_1^{\mathcal{P}} . \ldots . \sigma_n^{\mathcal{P}}$. In addition, from b) we know that for each path τ there exists some path $\tau_\phi = (g_1^\phi, \sigma_1^\phi).(g_2^\phi, \sigma_2^\phi) \ldots (g_n^\phi, \sigma_n^\phi)$. Consequently, the conditions (A) and (B) to be proved are respectively the following:

(A) $mgu(\sigma_1^{\mathcal{P}} \ldots \sigma_n^{\mathcal{P}}, \sigma_1^\phi \ldots \sigma_n^\phi) \neq \emptyset$.
(B) $g_1 \sqsubseteq (\sigma_1^{\mathcal{P}} \Vdash g_1^\phi)$ and $\forall 1 \leq j \leq n . \exists 1 \leq i \leq n$ such that $\sigma_1^{\mathcal{P}} . \ldots . \sigma_j^{\mathcal{P}}$ is a prefix of $\sigma_1 . \ldots . \sigma_i$ and $g_1 \bigwedge_{2 \leq l \leq i} (g_l) \Theta(\sigma_1 \ldots \sigma_{l-1})) \sqsubseteq (\sigma_j^{\mathcal{P}} \Vdash g_j^\phi) \Theta(\sigma_1^{\mathcal{P}} \ldots \sigma_{j-1}^{\mathcal{P}})$

To prove (A), according to c), we have to prove that For all $1 \leq i \leq n$ there exists $i \leq j \leq n$, such that:
$mgu(\Omega(\sigma_1^{\mathcal{P}}, \sigma_1^\phi, h_0) \ldots \Omega(\sigma_n^{\mathcal{P}}, \sigma_n^\phi, h_{n-1}), \sigma_1^\phi \ldots \sigma_n^\phi) \neq \emptyset$.

e) Since $h_n = \varepsilon$, we have: $\Omega(\sigma_1^{\mathcal{P}}, \sigma_1^\phi, h_0)) \ldots \Omega(\sigma_n^{\mathcal{P}}, \sigma_n^\phi, h_{n-1}) = \sigma_1^{\mathcal{P}} \ldots \sigma_n^{\mathcal{P}}$,
f) From the definitions of functions $\Vdash$ and Ω there exists a substitution Δ, such that: $\sigma_1^{\mathcal{P}} \ldots \sigma_n^{\mathcal{P}} = (\sigma_1^\phi \ldots \sigma_n^\phi)\ \Delta$, where $\sigma_n^\phi \neq \varepsilon$ (accepting or inserting case).
g) From e) and f), we get:
$mgu(\Omega(\sigma_1^{\mathcal{P}}, \sigma_1^\phi, h_0)) \ldots \Omega(\sigma_i^{\mathcal{P}}, \sigma_i^\phi, h_{i-1}), (\sigma_1^\phi \ldots \sigma_i^\phi)) \neq \emptyset$ which satisfies (A).

The proof of (B) is based on term reductions that are similar to those used in the proof of condition (C) in (I). For space limitation, this proof will not presented in the paper.

Hereafter, we formulate the transparency theorem stating that any possible execution of the original EFSM, that respects the security property, is a possible execution of the EFSM resulting from injection.

Theorem 2 (Transparency). [6] *Let P and $\mathcal{P}'$ be two programs specified by the EFSMs $E_{\mathcal{P}}$ and $E_{\mathcal{P}'}$ respectively. Let Φ a property enforced by the EA specified by the EFSM E_ϕ and $P \circlearrowleft \Phi$ be the program specified by the EFSM $E_{\phi 2 \mathcal{P}} = E_{\mathcal{P}} \circlearrowleft_{\mathcal{H},\mathcal{R}}^{\theta} E_\phi$. We have:*

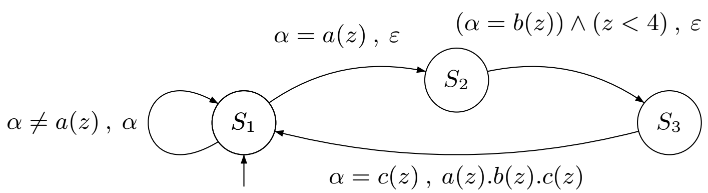
$$\textit{If } \mathcal{P}' \preceq \mathcal{P} \textit{ and } \mathcal{P}' \vDash \Phi \textit{ then } \mathcal{P}' \precsim \mathcal{P} \circlearrowleft \Phi$$

6 Example

In this section, we show through an example how we can use our approach to generate a program satisfying a property from one that did not satisfy it. In this example, we take an EFSM E_{ϕ_2} specifying the EA enforcing a property ϕ_2 (Figure 2) and an EFSM $E_{\mathcal{P}_1}$ representing some program $\mathcal{P}_1$ (Figure 3) not satisfying ϕ_2. Indeed, the sequences $a(x)c(x)$ that can be executed following the path $T_1 \xrightarrow{(x>1,a(x))} T_2 \xrightarrow{(x>=5,\varepsilon)} T_3 \xrightarrow{(y>0,c(x))} T_4$ are examples of traces that are not satisfying ϕ_2 and can result from executing $\mathcal{P}_1$. The effects of the four actions a(x),b(x),c(x), and d(x), involved in $E_{\mathcal{P}_1}$ and E_{ϕ_2} are the followings:

[5] Recall that any path can be considered as extension of itself.

[6] For space limitation, the proof of this theorem has not been presented.



$\alpha = a(z)\ ,\ \varepsilon$
$(\alpha = b(z)) \wedge (z < 4)\ ,\ \varepsilon$
S_2
$\alpha \neq a(z)\ ,\ \alpha$
S_1
S_3
$\alpha = c(z)\ ,\ a(z).b(z).c(z)$

Fig. 2. The EFSM Specifying the EA Enforcing the Property ϕ_2

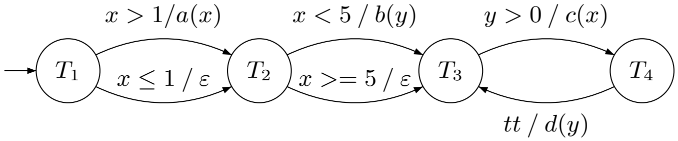
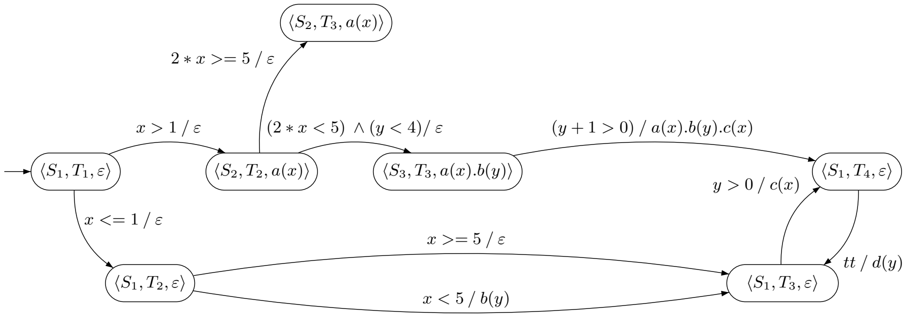


Fig. 3. The EFSM $\mathrm{E}_{\mathcal{P}_1}$



$\langle S_2, T_3, a(x)\rangle$
$2 * x >= 5\ /\ \varepsilon$
$x > 1\ /\ \varepsilon$
$(2 * x < 5)\ \wedge (y < 4)/\ \varepsilon$
$(y + 1 > 0)\ /\ a(x).b(y).c(x)$
$\langle S_1, T_1, \varepsilon\rangle$
$\langle S_2, T_2, a(x)\rangle$
$\langle S_3, T_3, a(x).b(y)\rangle$
$\langle S_1, T_4, \varepsilon\rangle$
$y > 0\ /\ c(x)$
$x <= 1\ /\ \varepsilon$
$x >= 5\ /\ \varepsilon$
$tt\ /\ d(y)$
$\langle S_1, T_2, \varepsilon\rangle$
$\langle S_1, T_3, \varepsilon\rangle$
$x < 5\ /\ b(y)$

Fig. 4. The EFSM Resulting From Injecting E_{ϕ_2} into E_{P_1}

- $\theta(a(x)) = \{x/2 * x\}$,
- $\theta(b(x)) = \{x/x + 1\}$,
- $\theta(c(x)) = \{x/x^3\}$,
- $\theta(d(x)) = \{x/x - 1\}$.

The injection of E_{ϕ_2} into E_{P_1} produces the EFSM depicted in Figure 4. After simplifying transitions by evaluating transitions guards, and by eliminating (1) transitions guarded by conditions that can be reduced to false, and (2) every path in which all the transitions actions are equal to ε, we got the optimized EFSM. Notice that in this new version of the resulting EFSM, the guards have been modified as follows:

- In the guard $2 * x < 5$, the action $a(x)$ has been feigned,
- In the guard $y + 1 > 0$, the action $b(y)$ has been feigned,

– The guard $y < 4$ is a dynamic test that guaranties that the argument of the action $b(y)$ will satisfy the condition required by the property ϕ_2.

7 Conclusion

In this paper, we present a rewriting-based formal approach for enforcing security properties. These properties are supposed to be enforceable by those EM that are specifiable by edit automata [9]. Being able to feign the execution of sensitive actions of controlled programs, edit automata are supposed to enforce a wide range of properties that cannot be enforced by conventional EM. However, feigning program actions will usually make the program behaving in discordance with its specification. The reason of this discordance is the fact that the effects of feigned actions are not reflected in the program states. In this paper, we highlighted this problem and showed how program rewriting can be a reliable enforcement alternative. The proposed formal framework used in our approach is based on EFSMs. EFSMs are expressive enough to specify edit automata as well as programs. We defined an injection operator that takes an EFSM specifying an edit automaton and embeds it into an EFSM representing a given program. The EFSM obtained from the injection is a secure version of the original program. We provided the formal results related to our operator as well as their corresponding proofs.

We are currently investigating the use of aspect oriented programming (AOP) [10] to implement EA-based program rewriting enforcement. The theoretical foundations of this ongoing task are based on the pointcut-advice model while its implementation is based on AspectJ [11]. One interesting thread of this work is building a catalogue of real-world security properties and the corresponding aspects enforcing them. Another future research thread related to the paper contribution is leveraging the existing state of the art of conformance testing for EA-based program rewriting enforcement. This thread is motivated by the fact that existing techniques are not adequate to deal with feigning actions and emulating their effects on rewritten programs. The expected techniques will generate the data required to test the conformance of an enforcement mechanism implementation to its formal specification.

References

1. Hamlen, K.W., Morrisett, G., Schneider, F.B.: Computability classes for enforcement mechanisms. ACM Trans. Program. Lang. Syst. 28(1), 175–205 (2006)
2. Schneider, F.B.: Enforceable security policies. ACM Transactions on Information and Systems Security 3(1), 30–50 (2000)
3. Erlingsson, U., Schneider, F.B.: Sasi enforcement of security policies: a retrospective. In: Proceedings of the New Security Paradigms Workshop, Caledon Hills, Ontario, Canada, pp. 87–95. ACM Press, New York (2000)
4. Evans, D., Twyman, A.: Flexible Policy-Directed Code Safety. In: IEEE Symposium on Security and Privacy, Oakland, California (May 1999)

5. Viswanathan, M.: Foundations for the Run-time Analysis of Software Systems. PhD thesis, University of Pennsylvania (2000)
6. Ligatti, J., Bauer, L., Walker, D.: Enforcing non-safety security policies with program monitors. In: di Vimercati, S.d.C., Syverson, P.F., Gollmann, D. (eds.) ESORICS 2005. LNCS, vol. 3679, pp. 355–373. Springer, Heidelberg (2005)
7. Ligatti, J., Bauer, L., Walker, D.: Edit automata: Enforcement mechanisms for run-time security policies. International Journal of Information Security 4(1-2), 2–16 (2005) (published online 26 October 2004)
8. Bauer, L., Ligatti, J., Walker, D.: More enforceable security policies. In: Foundations of Computer Security, Copenhagen, Denmark, July 25-26, pp. 95–104 (2002)
9. Ligatti, J., Bauer, L., Walker, D.: Enforcing non-safety security policies with program monitors. Technical Report TR-720-05, Princeton University (January 2005)
10. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: Akşit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
11. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.: An overview of aspectJ. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, p. 327. Springer, Heidelberg (2001)
12. Lamport, L.: Proving the correctness of multiprocess programs. IEEE Transactions of Software Engineering 3(2), 125–143 (1977)
13. Hamlen, K., Morrisett, G., Schneider, F.: Computability classes for enforcement mechanisms. Technical Report TR2003-1908, Cornell University (2003); To appear in ACM Transactions on Programming Languages and Systems
14. Necula, G.C.: Proof-carrying code. In: Conference Record of POPL 1997: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 1997, pp. 106–119 (1997)