Automatic Testing of Access Control for Security Properties*

Hervé Marchand, Jérémy Dubreil, and Thierry Jéron

INRIA, centre Rennes - Bretagne Atlantique, first.last@irisa.fr

Abstract. In this work, we investigate the combination of controller synthesis and test generation techniques for the testing of open, partially observable systems with respect to security policies. We consider two kinds of properties: integrity properties and confidentiality properties. We assume that the behavior of the system is modeled by a labeled transition system and assume the existence of a black-box implementation. We first outline a method allowing to automatically compute an ideal access control ensuring these two kinds of properties. Then, we show how to derive testers that test the conformance of the implementation with respect to its specification, the correctness of the real access control that has been composed with the implementation in order to ensure a security property, and the security property itself.

1 Introduction

There has been an increasing interest in research about computer security in the past decades. Indeed, the emergence of web services and the improvements of the possibilities of mobile and embedded systems allow lots of new and interesting features. But some of these services such as on-line payment, medical information storage or e-voting systems may deal with some critical information. In the meantime, having more applications and devices for accessing these services also increases the possibilities for such information to flow or to be erased/corrupted. To avoid security breaches, using automatic tools based on formal methods for security analysis can be beneficial. In this context, there has been a growing interest in verification [4, 13], active testing of security properties [8] or passive testing (supervision) [14]. In order to specify such automatic analysis methods, security properties are generally classified into three different categories [3]: *availability* (actions allowed by the security policy are always available), *integrity* (something illegal cannot be performed) and *confidentiality* (secret information cannot be inferred) [5]. We focus here on particular classes of confidentiality and integrity properties.

In this paper, we assume that the system can be modeled by a finite transition system labeled over an alphabet Σ. The communication interface between the system and a user (possibly an attacker) is given by a subalphabet $\Sigma_a \subseteq \Sigma$. The integrity properties we are considering are properties that can be expressed

* This work was partially supported by the Politess RNRT project.

by means of regular sets of execution trajectories in Σ^*. To describe the confidentiality properties, we adopt the formalism of [5]: the confidential information is the membership of trajectories to a secret given by a regular language, and the secret is said to be *opaque* whenever the attacker, partially observing the system, cannot infer that the current trajectory belongs to the secret. Note that the definition of opacity is general enough to model other notions of information flow like trace-based non-interference and anonymity (see [5]). Notice also that *secrecy* [1] can be handled as a particular case of opacity and thus our framework applies to secrecy as well.

In order to avoid security breaches (information flow or integrity violation), a possible solution consists in coupling the system with a monitor, in charge of detecting when confidential information has leaked (resp. integrity has been violated) or will leak (resp. will be violated). Assuming that monitors observe only a subset Σ_m of the actions of the system, necessary and sufficient conditions for the existence of monitors were obtained in [11]. In [9], we went one step further and provided techniques allowing to compute the least restrictive access control allowing to restrict the behavior of the system in order to avoid the violation of the properties. In this work, we assumed that the controller observes only a subset Σ_m of Σ, including all controllable actions, and in the case of confidentiality, we also requested the two alphabets Σ_m and Σ_a, the alphabet of the attacker, to be comparable.

In this paper, we investigate the problem of testing whether an implementation conforms to a security policy. We consider an implementation $\mathcal{I}$ of the system, a specification S and an implementation $\mathcal{I}_{AC}$ (composed with $\mathcal{I}$) in charge of ensuring the required security policy. To validate this implementation $\mathcal{I} \times \mathcal{I}_{AC}$ (having Σ_a as communication interface with the users), we adapt the classical conformance testing method by deriving testers that test both the security properties and the conformance of the implementation (as in [7] for observable safety properties[1]) to its specification, and also test the implemented access controls $\mathcal{I}_{AC}$. Our testing process is as follows: Step 1 concerns the automatic synthesis from S of a formal model C of an access control with respect to the security properties; Step 2 is a test generation algorithm that takes the specification, a security property and its corresponding access control, and produces a test case for checking the security property on the implementation and the implemented access control; finally, Step 3 is standard conformance test execution, which may detect the following inconsistencies:

- violation of the access control by the controlled implementation;
- violation of the security policy by the controlled implementation;
- violation of conformance of the implementation w.r.t. its specification

To generate the testers, we first adopt the attacker's point of view (i.e. the test execution is performed via the interface Σ_a) and then the administrator's point of view (whose communication interface may differ from Σ_a and Σ_m).

[1] In [7], the safety properties are only concerned with the observable behavior of the system and not with its internal behavior, as it is the case in this paper.

The structure of the document is as follows: In section 2, we define the mathematical terminology and notions used throughout the paper. In Section 3, we formalize the notions of confidentiality and integrity and outline the verification of these properties. Section 4 describes how to compute an access control that prevents the violation of an information flow or an integrity property. Finally, Section 5 is devoted to the presentation of our testing methodology.

2 Models & Notations

Let Σ be a finite alphabet of actions. A *string* is a finite-length sequence of actions in Σ and ε denotes the empty string. Given a string s. The set of all strings formed by actions in Σ is denoted by Σ^*. Any subset of Σ^* is called a *language* over Σ. Let L be a language over Σ. L is said to be *extension-closed* when $L.\Sigma^* = L$. The *prefix-closure* of L is defined as $\overline{L} = \{s \in \Sigma^* \mid \exists t \in \Sigma^* \text{ s.t. } st \in L\}$. A language L is said *prefix-closed* whenever $L = \overline{L}$.

We assume that the behaviors of systems are modeled by Labeled Transitions Systems (LTS for short). The formal definition of an LTS is as follows:

Definition 1 (LTS). *An LTS is a 4-tuple $S = (Q_S, \Sigma, \rightarrow_S, q_S^0)$ where Q_S is a finite set of states, Σ is the alphabet of* actions*, $q_S^0 \in Q_S$ is the initial state, and $\rightarrow_S \subseteq Q_S \times \Sigma \times Q_S$ is* the partial transition relation.

Notations. we consider a given LTS $S = (Q_{\mathrm{S}}, \Sigma, \rightarrow_{\mathrm{S}}, q_{\mathrm{S}}^0)$.

- We write $q \xrightarrow{a}_S q'$ for $(q, a, q') \in \rightarrow_S$ and $q \xrightarrow{a}_S$ for $\exists q' \in Q_S,\ q \xrightarrow{a}_S q'$. We extend $\rightarrow_{\mathrm{S}}$ to arbitrary sequences by setting: $q \xrightarrow{\varepsilon}_{\mathrm{S}} q$ for every state q, and $q \xrightarrow{s\sigma}_{\mathrm{S}} q'$ whenever $q \xrightarrow{s}_{\mathrm{S}} q''$ and $q'' \xrightarrow{\sigma}_{\mathrm{S}} q'$, for some $q'' \in Q_{\mathrm{S}}$.
- Given $\Sigma' \subseteq \Sigma$, S is said to be Σ'-*complete* whenever $\forall q \in Q_{\mathrm{S}}, \forall a \in \Sigma', q \xrightarrow{a}_{\mathrm{S}}$.
- We set for any language $L \subseteq \Sigma^*$ and any set of states $X \subseteq Q_{\mathrm{S}}$,

$$\Delta_{\mathrm{S}}(X, L) \triangleq \{q \in Q_{\mathrm{S}} \mid \exists s \in L, \exists q' \in X,\ q' \xrightarrow{s}_{\mathrm{S}} q\}.$$

  A set of states $X \subseteq Q_{\mathrm{S}}$ is said to be *stable* if $\Delta_{\mathrm{S}}(X, \Sigma^*) \subseteq X$.
- $L(S) = \{l \in \Sigma^*, {}^0_{\mathrm{S}} \xrightarrow{l}_{\mathrm{S}}\}$ denotes the set of trajectories of the system S. Given a set of marked states $F_{\mathrm{S}} \subseteq Q_{\mathrm{S}}$, the *marked language* is defined as $L_{F_{\mathrm{S}}}(S) = \{l \in \Sigma^* \mid \exists q \in F_{\mathrm{S}}, q_{\mathrm{S}}^0 \xrightarrow{l}_{\mathrm{S}} q\}$, i.e. the set of trajectories that may end in F_S.

We now define the parallel composition of two LTSs.

Definition 2 (Parallel composition). *Let $S^i = (Q^i, \Sigma^i, \rightarrow_{S^i}, q_{S^i}^0)$, $i = 1, 2$ be two LTSs. The* parallel composition *between S^1 and S^2 is an LTS $S^1 \times S^2 = (Q^1 \times Q^2, \Sigma^1 \cup \Sigma^2, \rightarrow_{S^1 \times S^2}, (q_{S^1}^0, q_{S^2}^0))$, where $\rightarrow_{S^1 \times S^2}$ is the smallest relation in $(Q^1 \times Q^2) \times (\Sigma^1 \cup \Sigma^2) \times (Q^1 \times Q^2)$ satisfying*

$$(q_1, q_2) \xrightarrow{\sigma}_{S^1 \times S^2} \begin{cases} (q_1', q_2') \text{ if } \sigma \in \Sigma^1 \cap \Sigma^2 \ \wedge\ q_1 \xrightarrow{\sigma}_{S^1} q_1' \ \wedge\ q_2 \xrightarrow{a}_{S^2} q_2' \\ (q_1', q_2) \text{ if } \sigma \in \Sigma^1 \setminus \Sigma^2 \wedge\ q_1 \xrightarrow{\sigma}_{S^1} q_1' \\ (q_1, q_2') \text{ if } \sigma \in \Sigma^2 \setminus \Sigma^1 \ \wedge\ q_2 \xrightarrow{\sigma}_{S^2} q_2' \end{cases}$$

Clearly, if $\Sigma^1 = \Sigma^2$, $L(S^1 \times S^2) = L(S^1) \cap L(S^2)$ and for sets of marked states $F_i \subseteq Q^i, i = 1, 2$, we have $L_{F_1 \times F_2}(S^1 \times S^2) = L_{F_1}(S^1) \cap L_{F_2}(S^2)$. If for $i = 1, 2$ the set F_i is stable in S^i, $F_1 \times F_2$ is stable in $S^1 \times S^2$.

Definition 3 (Completion). *Given an LTS $S = (Q, \Sigma, \rightarrow, q^0)$, a fresh state q_{new} and a subalphabet $\Sigma' \subseteq \Sigma$, the (Σ', q_{new})-completion of S is an LTS $Comp^{q_{new}}_{\Sigma'}(S) = (Q \cup \{q_{new}\}, \Sigma, \rightarrow', q^0)$, where*

$$\rightarrow' = \rightarrow \cup \{q \xrightarrow{a} q_{new} \mid q \in Q \cup \{q_{new}\}, a \in \Sigma' \text{ s.t. } \neg(q \xrightarrow{a})\}$$

Observable Behavior. The interface between a user and the system is specified by a subalphabet of actions Σ_a. The behavior that is visible by a user, is then defined by its *projection.* denoted by Π_{Σ_a} from Σ^* to Σ_a^* that erases in a sequence of Σ^* all actions not in Σ_a. Formally,

$$\begin{aligned} \Pi_{\Sigma_a} : \Sigma^* &\rightarrow \Sigma_a^* \\ \varepsilon &\mapsto \varepsilon \\ s\sigma &\mapsto \begin{cases} \Pi_{\Sigma_a}(s)\sigma \text{ if } \sigma \in \Sigma_a \\ \Pi_{\Sigma_a}(s) \text{ otherwise} \end{cases} \end{aligned}$$

This definition extends to any language $K \subseteq \Sigma^*$: $\Pi_{\Sigma_a}(K) = \{\mu \in \Sigma_a^* \mid \exists s \in K,\ \mu = \Pi_{\Sigma_a}(s)\}$. In particular, given an LTS S over Σ and a set of observable actions $\Sigma_a \subseteq \Sigma$, the set of *observed traces* of S is $\mathcal{T}_{\Sigma_a}(S) = \Pi_{\Sigma_a}(L(S))$. Conversely, given $K \subseteq \Sigma_a^*$, the *inverse projection* of K is $\Pi^{-1}_{\Sigma_a}(K) = \{s \in \Sigma^* \mid \Pi_{\Sigma_a}(s) \in K\}$. Given an observation trace μ of S, we define $[\![\mu]\!]_{\Sigma_a}$ as the set of trajectories of S *compatible* with μ. These are trajectories of S having trace μ. Formally:

$$[\![\mu]\!]_{\Sigma_a} \triangleq \Pi^{-1}_{\Sigma_a}(\mu) \cap L(S)$$

An LTS S is said to be *deterministic* if for all $q \in Q_S$, for all $a \in \Sigma$, $q \xrightarrow{a}_S q'$ and $q \xrightarrow{a}_S q''$ implies $q' = q''$. In the sequel, we will need to build, a deterministic LTS $Det_{\Sigma_a}(S)$ over the alphabet Σ_a preserving the set of traces of a (non-deterministic) LTS S, i.e. $L(Det_{\Sigma_a}(S)) = \mathcal{T}_{\Sigma_a}(S)$.

Definition 4 (Determinization). *Let $S = (Q_S, \Sigma, \rightarrow_S, q_S^0)$ be an LTS and $\Sigma_a \subseteq \Sigma$ the subalphabet of observable actions. The* determinization of S *with respect to Σ_a is the LTS $Det_{\Sigma_a}(S) = (\mathcal{X}, \Sigma_a, \rightarrow_d, X^0)$ where $\mathcal{X} = 2^{Q_S}$ (the set of subsets of Q_S called* macro-states*), $X^0 = \Delta_S(\{q_S^0\}, [\![\epsilon]\!]_{\Sigma_a})$ and $\rightarrow_d$ defined by the set $\{(X, a, \Delta_S(X, [\![a]\!]_{\Sigma_a}) \mid X \in \mathcal{X} \text{ and } a \in \Sigma_a\}$.*

With the above definition, $\Delta_{Det_{\Sigma_a}(S)}(X^0, \mu)$ consists of states reached from q_S^0 by trajectories in $[\![\mu]\!]_{\Sigma_a}$ in S.

3 Security Properties

In this section, we formalize two kinds of security properties, namely confidentiality (something secret cannot be revealed to an attacker) and integrity (an attacker cannot make the system evolve into some bad configurations). We also outline the verification of these properties on the specification of the system S.

3.1 Integrity property

We consider integrity properties that can be expressed as safety properties on the trajectories of the system. As usual we model their negation by an observer.

Definition 5 (Integrity). *The negation of an integrity property ψ is given by the marked language of a complete deterministic LTS $\bar{\psi} = (Q_{\bar{\psi}}, \Sigma, \rightarrow_{\psi}, q^0_{\bar{\psi}})$, with a stable set of accepting states $F_{\bar{\psi}}$. We denote $L_{\bar{\psi}} = L_{F_{\bar{\psi}}}(\bar{\psi})$.* ◇

Intuitively, the trajectories of S that belong to $L_{\bar{\psi}}$ are sequences that violate the integrity property. Note that the set of states $F_{\bar{\psi}}$ is stable, because, once the integrity property is violated, it is forever. The verification is as follows:

Definition 6. *Given a system S and an integrity property ψ, ψ is satisfied (noted $S \models \psi$) whenever $L_{F_{\bar{\psi}}}(\bar{\psi}) \cap L(S) = \emptyset$.*

If ψ is not satisfied by the system, the unique supremal sublanguage of $L(S)$ that satisfies ψ is regular and given by $L(S) \setminus L_{\bar{\psi}}$.

3.2 Confidentiality property

Consider an LTS S over Σ and $\Sigma_a \subseteq \Sigma$. The alphabet Σ_a defines the interface allowing a user to interact with S. We formalize a secret φ as follows:

Definition 7 (Secret). *A* secret *φ is defined by the marked language of a complete and deterministic LTS, $\varphi = (Q_{\varphi}, \Sigma, \rightarrow_{\varphi}, q^0_{\varphi})$ with a set of accepting states $F_{\varphi} \subseteq Q_{\varphi}$. We denote $L_{\varphi} = L_{F_{\varphi}}(\varphi)$.* ◇

The language L_{φ} represents the confidential information on the execution of S. Now a user is considered as an attacker ($\mathcal{A}$) willing to catch this confidential information. It is armed for this with full information on the structure of S but only partial observation upon its behavior, namely the observed traces in Σ_a^*. This leads to the definition of opacity adapted from [5] for secret predicates given as regular languages.

Definition 8 (Opacity). *Given a system S, a secret φ is said to be* opaque *w.r.t. $L(S)$ and the interface Σ_a if for all $s \in L(S), [\![\Pi_{\Sigma_a}(s)]\!]_{\Sigma_a} \not\subseteq L_{\varphi}$* ◇

Intuitively, a secret is opaque whenever every secret sequence s of the system is observationally equivalent to at least one non secret sequence s'. Equivalently, L_{φ} is opaque w.r.t. $L(S)$ and Σ_a if and only if for all $\mu \in \mathcal{T}_{\Sigma_a}(S)$, $[\![\mu]\!]_{\Sigma_a} \not\subseteq L_{\varphi}$.

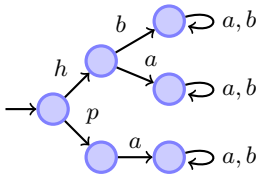


Fig. 1. An example of interference

Example 1. Let S be the LTS of Figure 1 with $\Sigma = \{h, p, a, b\}$, $\Sigma_a = \{a, b\}$. The secret, that should not be revealed is the occurrence of the (unobservable) action h (namely, $L_\varphi = \Sigma^* h \Sigma^*$). L_φ is not opaque w.r.t. S and Σ as the sole compatible sequence with the trace b is $h.b \in L_\varphi$. □

The definition of opacity extends to a family of languages $\mathcal{L} = \{L_1, L_2, \cdots, L_k\}$: the secret $\mathcal{L}$ is *opaque* with respect to S and Σ_a if for all $L \in \mathcal{L}$, for all $s \in L(S)$, $[\![\Pi_{\Sigma_a}(s)]\!]_{\Sigma_a}) \not\subseteq L$. Thus within our framework, it is also possible to express other confidentiality properties. For example, [1] introduced the notion of *secrecy* of a language L_φ. Intuitively, L_φ is *secret* w.r.t. S and Σ_a whenever after any observation μ, the attacker neither knows that the current execution is in L_φ nor in $L(S) \setminus L_\varphi$. Secrecy can thus be handled considering the opacity w.r.t. to the family $\{L_\varphi, L(S) \setminus L_\varphi\}$. This notion is suitable when the secret concerns the value of some variables.

Remark 1. All the results presented in this paper for opacity with a single secret can be extended to a family of secrets and thus to secrecy.

The following proposition gives a necessary and sufficient condition for opacity.

Proposition 1. *Given a system $S = (Q_s, \Sigma, \rightarrow_s, q_s^0)$, a secret $\varphi = (Q_\varphi, \Sigma, \rightarrow_\varphi, q_\varphi^0)$ equipped with $F_\varphi \subseteq Q_\varphi$, and an interface $\Sigma_a \subseteq \Sigma$, φ is opaque w.r.t. S and Σ_a if and only if $L_F(Det_{\Sigma_a}(S \times \varphi)) = \emptyset$, with $F = 2^{Q_S \times F_\varphi}$.*

It is easy to see that $L_F(Det_{\Sigma_a}(S \times \varphi))$ is the set of observations traces μ such that all trajectories in S compatible with μ fall into $L_{F_\varphi}(\varphi)$, i.e. the set of observations for which the attacker $\mathcal{A}$ knows that the current execution reveals φ. Hence, checking the opacity of φ consists in checking that this language is empty. This can be done by checking that F is not reachable in $Det_{\Sigma_a}(S \times \varphi)$.

Remark 2. Let S_1 and S_2 be two LTSs acting upon Σ, a secret φ such that $L_\varphi \subseteq \Sigma^*$ and an interface Σ_a. If φ is opaque w.r.t. $L(S_1)$ and $L(S_2)$, then it is opaque w.r.t. $L(S_1) \cup L(S_2)$, but not necessarily w.r.t. $L(S_1) \cap L(S_2)$. Given three LTSs S_1, S_2 and S_3 acting upon Σ such that $L(S_1) \subseteq L(S_2) \subseteq L(S_3)$, φ may be opaque w.r.t. $L(S_2)$ but not opaque w.r.t. $L(S_1)$ or $L(S_3)$.

When φ is not opaque w.r.t. $L(S)$ and Σ_a, it may be still possible to restrict the behavior of S so that L_φ becomes opaque. This is the aim of the next proposition.

Proposition 2 ([2]). *Given a system S and a secret φ, the supremal prefix-closed sub-language $L' \subseteq L(S)$, s.t φ is opaque w.r.t. L' is regular and given by*

$$\text{OP}^\uparrow(L(S), L_\varphi, \Sigma_a) = L(S) \setminus ((L(S) \setminus \Pi_{\Sigma_a}^{-1}(\Pi_{\Sigma_a}(L(S) \setminus L_\varphi))).\Sigma^*) \qquad (1)$$

Intuitively, the language $\Pi_{\Sigma_a}^{-1}(\Pi_{\Sigma_a}(L(S) \setminus L_\varphi))$ is the set of "safe" trajectories that do not reveal L_φ, whereas any trajectory in its complement $L(S) \setminus \Pi_{\Sigma_a}^{-1}(\Pi_{\Sigma_a}(L(S) \setminus L_\varphi))$ reveals L_φ (as well as any extension with a sequence in Σ^*, because, once L_φ has been revealed, this holds forever). Complementing again gives the supremal language $\text{OP}^\uparrow(L(S), L_\varphi, \Sigma_a)$.

4 Automatic synthesis of an Access Control

In this section, we propose to enforce opacity or integrity by supervisory control which consists in restricting the system behavior by pairing it with an access control. For implementability reasons, conditions on the admissible restrictions of $L(S)$ have to be put (controllability and normality). We assume that the interface of the controller is $\Sigma_m \subseteq \Sigma$. We will explicit the conditions under which the most permissive opacity (resp. integrity) control can be computed. The next section introduces a few notions of supervisory control theory.

4.1 Supervisory control theory overview

Given a prefix-closed behavior $K \subseteq L(S) \subseteq \Sigma^*$ expected from the system S, the goal of supervisory control is to enforce this behavior on S by pairing this system with a monitor (also called controller) modeled by an LTS $C = (Q_C, \Sigma_m, \rightarrow_C, q_C^0)$ that observes a subset Σ_m of the actions in Σ and controls a subset Σ_c of the actions in Σ, i.e. enables or disables each instance of these controllable actions. $\Sigma \setminus \Sigma_c$ is the set of uncontrollable actions. $\Sigma \setminus \Sigma_m$ is the set of unobservable actions. We now recall some basic concepts of supervisory control theory [6].

Definition 9 (Controllability and Normality). *A prefix-closed language $K \subseteq L(S)$ is*

- controllable *w.r.t.* $L(S)$ *and* Σ_c *if* $K.(\Sigma \setminus \Sigma_c) \cap L(S) \subseteq K$.
- Normal *w.r.t.* $L(S)$, Σ_m *if* $K = \Pi_{\Sigma_m}^{-1}(\Pi_{\Sigma_m}(K)) \cap L(S)$,

Controllability means that no uncontrollable action needs to be disabled to exactly confine the system $L(S)$ to K. Normality means that K can be exactly recovered from its projection and from S. Under the assumption $\Sigma_c \subseteq \Sigma_m$, there exists a supremal controllable and normal prefix-closed sub-language $K^\uparrow$ of K corresponding to the largest language included in K that can be enforced by control and this language is regular. If not empty, there exists a controller C, said *maximal* such that $L(C \times S) = K^\uparrow$. Example 2 in Section 4.2 illustrates the computation of this controller ([6] for a more complete review of the control theory of discrete event systems).

4.2 Access Control Synthesis

In the sequel, we assume that an attacker has full knowledge of the structure of S, that he knows the controller's interface Σ_m and is able to perform all the computations made by the administrator to compute the controller. In particular, the attacker knows that the controller is maximal and since there is only one optimal controller, the structure of the controller can be deduced by the attacker. In the rest of the paper, it is always assumed that $\Sigma_c \subseteq \Sigma_m$ (the controllable actions are observed by the controller). Next, we outline the methodologies allowing to compute access controls for ensuring a confidentiality property and then for ensuring an integrity property.

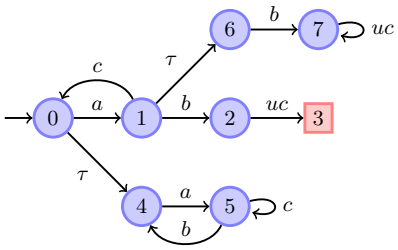
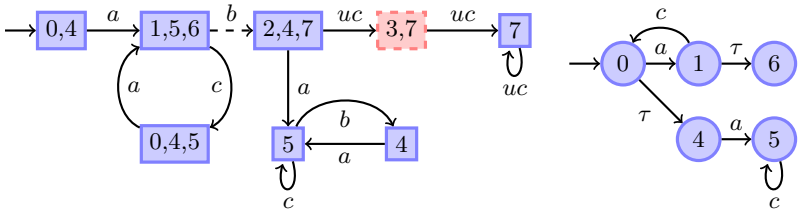


Fig. 2. $S \times \bar{\psi}$

Ensuring an Integrity property. Given a system S and an integrity property ψ, modeled by the negation of a safety property $\bar{\psi}$, the aim is to compute (when it exists) the maximal controller $C = (Q_C, \Sigma_m, \rightarrow_C, q_C^0)$ such that $C \times S \models \psi$. For such a property, the solution can be simply obtained using the control theory presented in the previous section by computing the greatest controllable and observable sublanguage of $L(S) \setminus L_{\bar{\psi}}$. The following example illustrates the computation of this controller.

Example 2. Figure 2 describes the product $S \times \bar{\psi}$ in which the state 3 is marked. Hence, a sequence that reaches the state 3 is violating ψ. We assume that $\Sigma_m = \{a, b, c, uc\}$ and that $\Sigma_c = \{a, b, c\}$. The controller decisions are performed according to the observed behavior of the system (depicted in Figure 3(a)). If the controller observes a sequence in $a.(c.a)^*.b.uc$, then he knows that the system is either in state 3 or 7. Thus, in order to avoid the state $\boxed{3,7}$, the controller needs to disable the event b (uc being uncontrollable). The obtained LTS (only keeping the accessible part) is the maximal controller such that $C \times S \models \psi$. The behavior of the controlled system is given in Figure 3(b). ◇

Ensuring a confidentiality property. Given a system S and a secret φ, our purpose is to decide whether there exists a maximally permissive controllable and observable access control $C = (Q_C, \Sigma, \rightarrow_C, q_C^0)$ such that φ is opaque w.r.t. $L(S \times C)$ and then to compute this controller C. We first illustrate the approach through an example.



(a) $det_{\Sigma_m}(S \times \bar{\psi})$ (b) $C \times S$

Fig. 3. Ensuring an integrity property by control

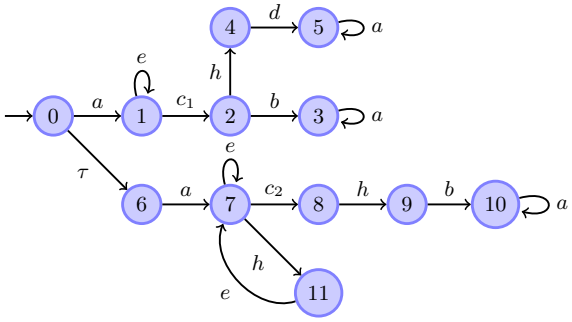
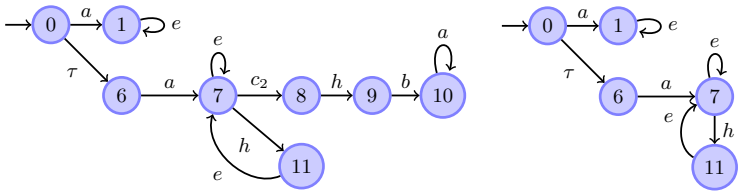


Fig. 4. Control of non-opacity (I)

Example 3. The system to be controlled is given in Figure 4. We assume that $\Sigma_a = \{a, b, d, e\}$, $\Sigma_m = \{a, c_1, c_2, b, d, e\}$, *and* $\Sigma_c = \{b, c_1, c_2, e\}$. *The secret is given by the language* $L_\varphi = \Sigma^*.h.\Sigma^*$. *When observing* d, *the attacker knows that* h *occurred and the secret is revealed (in state 5). By control, action* c_1 *is disabled, thus avoiding the uncontrollable sequence* $h.d$ *to be triggered, and the LTS depicted in Figure 5(a) is obtained. However, doing so, the secret is now*



(a) Step 1 (b) Step 2

Fig. 5. Control of non-opacity (II)

revealed to the attacker who knows the control law after the observation of the action b, *which leads to disable the action* c_2, *giving the LTS of Figure 5(b). The secret is now opaque with respect to this LTS, which is the maximal sub-LTS of the system with this property.* ◇

We now give sufficient conditions under which such a maximal access control exists. For details, please refer to [9, 10].

Theorem 1. *[9, 10] Let* S *be a system and* φ *a secret. Under the assumption* $\Sigma_c \subseteq \Sigma_m$, *there exists a maximal access control* C *such that* φ *is opaque w.r.t.* $L(S \times C)$ *and* Σ_a *whenever*

- φ *is opaque w.r.t.* $L(S) \cap \Sigma_{uc}^*$ *and* Σ_a,
- *(1)* $\Sigma_m \subseteq \Sigma_a$, *or (2)* $\Sigma_a \subseteq \Sigma_m$.

Assumption (1) means that the attacker's observation is better than the controller's one. In this case the controlled system can be obtained by computing $\text{OP}^{\uparrow}(S, \varphi, \Sigma_a)$ and the supremal controllable and observable sub-language of this language. Assumption (2) means that all actions of the attacker are observable by the controller, but only a subset is controllable. One can imagine a firewall for Internet services where the controller can filter out the requests sent by the attacker to the system, whereas the outputs of the system cannot be disabled by the controller. In this case, a novel algorithm has been introduced to compute the controller (details can be found in [9, 10]).

5 Automatic test generation for security policies

In this section, we propose to test whether an implementation of the system behaves as expected with respect to its specification model, and with respect to the security properties and the access control computed in the preceding section. We adapt the conformance testing framework whose aim is to establish whether a black-box implementation conforms to its specification model.

We consider an implementation $\mathcal{I}$ of the system[2], a specification model S and a security property (either a confidentiality property for which the secret is given by φ or an integrity property ψ). We assume that an access control C with interface Σ_m has been computed for S in order to ensure one of these security properties at the specification level, and that an implementation of this access control $\mathcal{I}_{AC}$ has been connected to $\mathcal{I}$ (we thus have $L(\mathcal{I} \times \mathcal{I}_{AC}) \subseteq L(\mathcal{I})$) with the same interface.

We consider the test architecture depicted in Fig. 6, in particular the tester is considered as an attacker with interface Σ_a, and we make explicit the difference between inputs and outputs of the system by partitioning the set of observable actions as $\Sigma_a = \Sigma_? \cup \Sigma_!$, where $\Sigma_?$ are inputs and $\Sigma_!$ are outputs.

The aim of the test campaign is then to check that the implemented access control $\mathcal{I}_{AC}$ reduces the behavior of $\mathcal{I}$ such that the security property is ensured. Hence, test execution is expected to detect the following inconsistencies:

- invalidation of the access control specification C by the controlled implementation,
- violation of the security property by the controlled implementation,
- violation of conformance between the implementation $\mathcal{I}$ and its specification S. The chosen conformance relation is $\leq_{io}$, a version of IOCO [15] in which

[2] As usual we consider that the implementation behaves like a model.

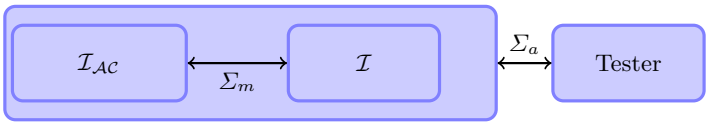


Fig. 6. Tester Architecture

quiescence is not taken into account. $\leq_{io}$ is formally defined by:

$$\mathcal{I} \leq_{io} S \stackrel{\Delta}{=} T_{\Sigma_a}(S).\Sigma_! \cap T_{\Sigma_a}(\mathcal{I}) \subseteq T_{\Sigma_a}(S) \quad (2)$$

Intuitively, after any trace of the specification and implementation, all outputs of the implementation must be specified.

We propose a test generation algorithm, which takes a specification S and a security property φ (or ψ) and its corresponding synthesized access control C, and produces a test case that, when executed on an implementation, attempts to push the implementation first into invalidating the implemented access control and second into violating the security property.

It is worthwhile noting that the properties on which the tests will be based do not only concern the observable behavior, but also the internal behavior. Thus assumptions are needed that bind the internal behavior of the implementation to the one of the specification. Hence, in the sequel, we shall assume that:

Assumption 1 : $\mathcal{I} \leq_{io} S \Rightarrow L(\mathcal{I}) = L(S)$

This assumption means that the behavior of the implementation corresponds to the one of the specification as far as the test campaign does not reveal any non conformance. Even though relatively restrictive, this assumption is necessary if one wants to test the implemented access control. Indeed, if $L(\mathcal{I})$ differs from $L(S)$, then the access control plugged on $\mathcal{I}$ would be different from the one on S. Remember (Remark 2) that opacity is not preserved by inclusion. Thus, if we assume that, for example, $L(\mathcal{I}) \subseteq L(S)$, then there could exist in $\mathcal{I}$ some information flows not present in S and reciprocally, which entails a different mechanism to enforce opacity.

Remark 3. If the aim of the test campaign was to discover whether an information flow exists (and if we were not interested in testing the access control itself), then Assumption 1 could be relaxed and become:

Assumption 2 : $\forall \mu \in T_{\Sigma_a}(\mathcal{I}) \cap T_{\Sigma_a}(S), \Pi^{-1}_{\Sigma_a}(\mu) \cap L(\mathcal{I}) \subseteq L(S)$

This assumption means that whenever an observation trace of the implementation is accepted by the specification, then all the sequences of the implementation compatible with this observation are also sequences of the specification. In particular, it entails that, if the secret is revealed in S after a trace μ, then it would be surely revealed in the implementation. ◇

In the sequel we shall assume that assumption 1 holds.

5.1 Computation of the Canonical Tester

In the following sections we focus on the automatic test generation for confidentiality properties (the methodology for integrity properties would be similar).

We first build a *Canonical Tester* which is the most general tester that can detect the inconsistencies described above. In the following section we will then describe how to select some test cases from this canonical tester.

As previously mentioned, we want to test the confidentiality property as well as the access control that has been plugged with the implementation in order to ensure confidentiality. The tester will thus be derived from the specification $S = (Q_{\mathrm{S}}, \Sigma, \rightarrow_{\mathrm{S}}, q^0_{\mathrm{S}})$, the secret $\varphi = (Q_\varphi, \Sigma, \rightarrow_\varphi, q^0_\varphi)$ equipped with F_φ, and the system controlled by the access control computed previously $S_C = S \times C = (Q_{\mathrm{SC}}, \Sigma, \rightarrow_{\mathrm{SC}}, q^0_{\mathrm{SC}})$. S_C specifies a safety property, the largest observable and controllable safety property included in $L(S)$ which guaranties that the secret φ is not leaked. $Comp^{V_{AC}}_{\Sigma}(S_C)$ equipped with the marked state V_{AC} is then an observer recognizing the negation of this property. Let us first consider the following LTS:

$$S^C_\varphi = (Q^C_\varphi, \Sigma, \rightarrow^C_\varphi, q^C_{0_\varphi}) = S \times \varphi \times Comp^{V_{AC}}_{\Sigma}(S_C)$$

S^C_φ can be equipped with the two following sets of marked states

- $F = Q_S \times F_\varphi \times (Q_{\mathrm{C}} \cup \{V_{AC}\})$;
- $F_{AC} = Q_S \times Q_\varphi \times \{V_{AC}\}$.

Let us describe some properties of S^C_φ. As both φ and $Comp^{V_{AC}}_{\Sigma}(S_C)$ are complete, we get $L(S^C_\varphi) = L(S)$. Moreover, $L_F(S^C_\varphi) = L(S) \cap L_\varphi \subseteq L_\varphi$ whereas $(L(S^C_\varphi) \setminus L_F(S^C_\varphi)) \cap L_\varphi = \emptyset$ and $L_{F_{AC}}(S^C_\varphi) \cap L(S_C) = \emptyset$. The role of F is to recognize trajectories of S satisfying the secret, while the role of F_{AC} is to recognize trajectories of S that violate the access control.

Example 4. Back to Example 3, the LTS S^C_φ is represented in Figure 7 with the rules that the square states belong to F_{AC} whereas the dashed square states belong to F. ◇

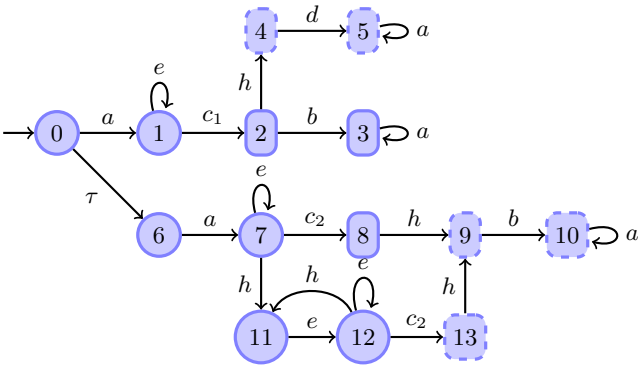


Fig. 7. S^C_φ.

A canonical tester for testing the conformance with respect to S is usually defined by $Test(S) = Comp^{Fail}_{\Sigma_!}(Det_{\Sigma_a}(S))$ [7]. As our aim is also to test the access control and confidentiality, the canonical tester is here built from S^C_φ as follows:

$$Test(S,\varphi) = (X, \Sigma_a, \rightarrow_t, X_o) = Comp^{Fail}_{\Sigma_!}(Det_{\Sigma_a}(S^C_\varphi))$$

$Test(S,\varphi)$ can be seen as a refinement of $Test(S)$. In fact, besides the detection of the non conformance of the implementation, $Test(S,\varphi)$ can also be used to detect the information flow induced by the secret φ as well as an incorrect implementation of the access control. But due to the test architecture, in particular the fact that the tester partially observes the system through Σ_a, the tester's verdicts are not given to trajectories but to observation traces, thus identifying all trajectories with the same observation.
For an observation $\mu \in \mathcal{T}(\mathcal{I} \times \mathcal{I}_{AC}) \cap L(Test(S,\varphi))$, the following verdicts are attached to $Test(S,\varphi)$:

$$\mathcal{O}_{Test(S,\varphi)}(\mu) = \begin{cases} NotConf & \text{if } \Delta_{Test(S,\varphi)}(X_0,\mu) = \{Fail\} \\ Leak & \text{if } \Delta_{Test(S,\varphi)}(X_0,\mu) \subseteq F \\ Violate_{AC} & \text{if } \Delta_{Test(S,\varphi)}(X_0,\mu) \subseteq F_{AC} \wedge \Delta_{Test(S,\varphi)}(X_0,\mu) \not\subseteq F \end{cases}$$

The interpretation of the verdicts is as follows:

- If $\mathcal{O}_{Test(S,\varphi)}(\mu) = NotConf$, then by definition of $Comp^{Fail}_{\Sigma_!}(Det_{\Sigma_a}(S^C_\varphi))$, $\exists \mu' \in \Sigma^*, a \in \Sigma_!, \mu = \mu'.a$ with $\mu' \in L(Det_{\Sigma_a}(S^C_\varphi)) = \mathcal{T}_{\Sigma_a}(S^C_\varphi) = \mathcal{T}_{\Sigma_a}(S)$, which entails that $\mu \in \mathcal{T}_{\Sigma_a}(S).\Sigma_!$, and $\mu \in \mathcal{T}_{\Sigma_a}(\mathcal{I} \times \mathcal{I}_{AC}) \subseteq \mathcal{T}_{\Sigma_a}(\mathcal{I})$, but $\mu \notin L(Det_{\Sigma_a}(S^C_\varphi)) = \mathcal{T}_{\Sigma_a}(S)$. Thus by definition of $\leq_{io}$, $\neg(\mathcal{I} \leq_{io} S)$ and the test campaign can be stopped.
- If $\mathcal{O}_{Test(S,\varphi)}(\mu) = Leak$, it means that $[\![\mu]\!]_a \subseteq L_F(S^C_\varphi)$ thus $[\![\mu]\!]_a \subseteq L_\varphi$, which entails that there is an information flow and the access control is not well implemented.
- If $\mathcal{O}_{Test(S,\varphi)}(\mu) = Violate_{AC}$, it means that $[\![\mu]\!]_{\Sigma_a} \subseteq L_{F_{AC}}(S^C_\varphi)$. But as $L_{F_{AC}}(S^C_\varphi) \cap L(S_C) = \emptyset$, we get $[\![\mu]\!]_{\Sigma_a} \cap L(S_C) = \emptyset$.
  Now, as $[\![\mu]\!]_{\Sigma_a} \not\subseteq L_{F_{AC}}(S^C_\varphi)$, there exists $s \in [\![\mu]\!]_{\Sigma_a} \setminus L_\varphi$. This implies that the access control is not well implemented, but the secret is not revealed so far.

Example 5. Back to example 4, The tester $Test(S,\varphi)$ is given by the LTS of Figure 8. We here assume that $\Sigma_a = \{a,b,d,e\}$ with $\Sigma_! = \{b,e\}$ and $\Sigma_? = \{a,d\}$. The verdict $Violate_{AC}$ is emitted if the state $\boxed{3,10}$ *is reached, meaning that the access control is not well implemented. If the sequence a.d is observed, the secret is revealed and the tester emits the verdict Leak (the state* $\boxed{5}$ *belongs to F). The verdict Fail is emitted whenever the Fail state is reached (the output "e" should not be observed when the tester is either in state* $\boxed{0,6}$ *or* $\boxed{3,10}$*).* $\diamond$

Remark 4. **Test from an administrator point of view.** *If the test campaign is performed via the interface $\Sigma'_m = \Sigma'_! \cup \Sigma'_?$ of an administrator $\mathcal{M}$ and does*

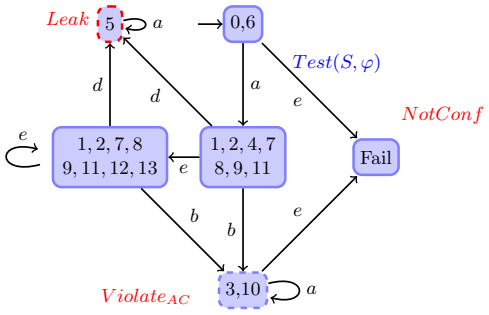


Fig. 8. $Test(S, \varphi)$.

not use the interface of the users, we need to adapt the computation of the tester in order to take into account the difference of observations [11].

The main difference concerns the computation of the verdict Leak. Indeed, the secret φ is revealed to the attacker by an execution $s \in L(S)$ if and only if $\Pi_a(s) \in L_F(Det_{\Sigma_a}(S \times \varphi))$. In other words, we are interested in testing the property: "The secret φ has been revealed to the attacker", which corresponds to the extension-closed language: $\Pi_a^{-1}(L_F(Det_{\Sigma_a}(S \times \varphi))) \cdot \Sigma^*$. This language can be recognized by an LTS Ω, equipped with a set of final states F_Ω such that:

$$\mathcal{L}_{F_\Omega}(\Omega) = \Pi_a^{-1}(L_F(Det_{\Sigma_a}(S \times \varphi))) \cdot \Sigma^* \tag{3}$$

Further, the computation of the tester $Tester_{\mathcal{M}}$ can simply be done by replacing φ by Ω in the test generation algorithm that we just described. $\diamond$.

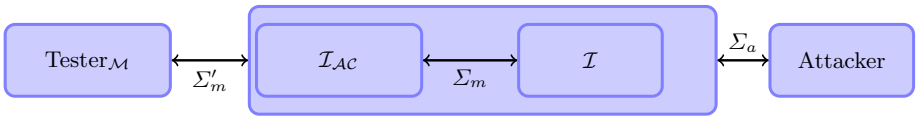


Fig. 9. Tester Architecture (II)

5.2 Test Selection

This operation is useful to target the test cases with respect to some particular behavior of the systems under test. For example, if we want to discover information flow, this operation consists in suppressing from $Test(S, \varphi)$ the subgraphs that cannot lead to some accepting set of states (e.g. F[3]). In that case, the main goal of testing is to check the violation of the opacity property after a trace of the specification and, if an implementation leads a tester (extracted from the specification) into a subgraph that cannot lead to *Leak*, the current test experiment will never be able to achieve this goal and it could be interesting to stop the test campaign. It that particular case, a new verdict *Inconclusive* is emitted.

[3] The selection can be performed with any combination of accepting states, depending on what the test campaign is focused on.

There are two situations, depending on whether the subgraph (from which *Leak* is unreachable) was entered through an input or an output:

- the subgraph has been entered by an *input*. In this case, the transition labeled by that input (together with the whole subgraph) are removed. Intuitively, the tester has control over this action, thus, it may decide not to stimulate the implementation with such input if it is sure that this will never lead to a *Leak* verdict.
- the subgraph has been entered by an *output* (that does not directly lead to $Fail$). In this case, only the transition labeled by that action is kept (the rest of the graph is removed). The destination of the transition is set to a new state called $Inconc$, which means that no *Leak* verdict can be given any more (but the conformance was not violated). Hence, for completeness, in this situation the verdict *Inconclusive* is emitted, i.e.
  $\forall \mu \in \mathcal{T}_{\Sigma_a}(\mathcal{I} \times \mathcal{I}_{AC}) \cap L(Test(S, \varphi))$,

$$\mathcal{O}_{Test(S,\varphi)}(\mu) = Inconclusive \text{ if } \Delta_{Test(S,\varphi)}(X_0, \mu) \subseteq \{Inconc\}.$$

Example 6. Applying this operation to the tester $Test(S, \varphi)$ of Figure 8 leads to the LTS depicted in Figure 10. After the reception of a "a" from the implementation, if the tester observes the output "b", then the tester knows that the secret cannot be revealed anymore, and the test campaign can be stopped.

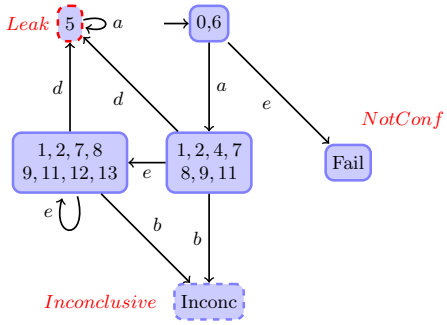


Fig. 10. The new tester $Test(S, \varphi)$.

6 Conclusion

In this work, we have been interested in the automatic test generation for security properties on partially observed systems by using the controller synthesis methodology to drive the test. Adopting a model-based approach, we assumed the existence of a specification of the system modeled by a finite transition system as well as the existence of a black-box implementation. We focused on two kinds of properties: integrity (safety property) and confidentiality (opacity property). The first step of our method consists in automatically computing access controls ensuring these two kinds of properties on the specification. We then

show how to derive testers that not only test the conformance of the implementation with respect to its specification, but also the security property itself as well as the correctness of the implemented access control composed with the implementation in order to ensure the security policy.

An interesting extension of this work would be to consider more expressive models mixing control and data. For these infinite models, the problem is that the computations of the controller and the tester rely on approximate analyses. This leads to investigate control and test techniques for security properties using the abstract interpretation theory. It would also be important to define the exact knowledge of the attacker in a case where computing the set of all behaviors may be impossible, as tackled by [12] for non-interference. Another issue would be to relax the hypothesis (1) linking internal behaviors of the implementation and the specification, while preserving the soundness of verdicts.

References

1. R. Alur, P. Černý, and S. Zdancewic. Preserving secrecy under refinement. In *ICALP '06: Proceedings (Part II) of the 33rd International Colloquium on Automata, Languages and Programming*, pages 107–118. Springer, 2006.
2. E. Badouel, M. Bednarczyk, A. Borzyszkowski, B. Caillaud, and P. Darondeau. Concurrent secrets. *Discrete Event Dynamic Systems*, 17:425–446, 2007.
3. M. Bishop. *Introduction to computer security.* Addison-Wesley Professional, 2004.
4. B. Blanchet, Abadi; M., and C. Fournet. Automated Verification of Selected Equivalences for Security Protocols. In *20th IEEE Symposium on Logic in Computer Science (LICS 2005)*, pages 331–340, Chicago, IL, June 2005.
5. J.W. Bryans, M. Koutny, L. Mazaré, and P. Ryan. Opacity generalised to transition systems. *International Journal of Information Security*, 7(6):421–435, May 2008.
6. C. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems.* Kluwer Academic Publishers, 1999.
7. C. Constant, T. Jéron, H. Marchand, and V. Rusu. Integrating formal verification and conformance testing for reactive systems. *IEEE Transactions on Software Engineering*, 33(8):558–574, August 2007.
8. V. Darmaillacq, J.-C. Fernandez, R. Groz, L. Mounier, and J.-L. Richier. Test generation for network security rules. In *TestCom 2006*, vol. 3964 of *LNCS*, 2006.
9. J. Dubreil, Ph. Darondeau, and H. Marchand. Opacity enforcing control synthesis. In *Workshop on Discrete Event Systems, WODES'08*, Gothenburg, Sweden, 2008.
10. J. Dubreil, Ph. Darondeau, and H. Marchand. Supervisory control for opacity. Technical Report 1921, IRISA, February 2009.
11. J. Dubreil, T. Jéron, and H. Marchand. Monitoring information flow by diagnosis techniques. In *European Control Conference (ECC)*, August 2009.
12. R. Giacobazzi and I. Mastroeni. Abstract non-interference: parameterizing non-interference by abstract interpretation. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 186–197. ACM, 2004.
13. G. Lowe. Towards a completeness result for model checking of security protocols. *Journal of Computer Security*, 7(2-3):89–146, 1999.
14. F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.
15. J. Tretmans. Testing concurrent systems: A formal approach. In *Concurrency Theory (CONCUR'99)*, number 1664 in LNCS, pages 46–65, 1999.