# Formalization of Security Properties: Enforcement for MAC Operating Systems and Verification of Dynamic MAC Policies

Jérémy Briffaut, Jean-François Lalande, Christian Toinard
Laboratoire d'Informatique Fondamentale d'Orléans
ENSI de Bourges – Université d'Orléans
88 bd Lahitolle, 18020 Bourges cedex, France
{jeremy.briffaut,jean-francois.lalande,christian.toinard}@ensi-bourges.fr

***Abstract*—Enforcement of security properties by Operating Systems is an open problem. To the best of our knowledge, the solution presented in this paper[1] is the first one that enables a wide range of integrity and confidentiality properties to be enforced. A unified formalization is proposed for the major properties of the literature and new ones are defined using a Security Property Language. Complex and precise security properties can be defined easily using the SPL language but the system includes 13 predefined protection templates. Enforcement of the requested properties is supported through a compiler that computes all the illegal activities associated with an existing MAC policy. Thus, we provide a SPLinux kernel that enforces all the requested Security Properties. When the MAC policies are dynamic, it becomes difficult to compute all the possible illegal activities. Our paper proposes a solution to that problem. A meta-policy includes evolution constraints for the MAC policies. A verification tool computes the requested security properties that are defined through the SPL language. That tool provides the illegal activities for all the possible MAC dynamic policies. Thus, the security administrator can verify the MAC policies and can optimize the requested security properties. It helps him to evaluate the memory and processor load associated with the enforcement of the request security properties. Efficiency is presented for protecting high-interaction honeypots.**

***Index Terms*—Security Properties, Protection, Mandatory Access Control, Verification**

## I. INTRODUCTION

Security of the Operating Systems usually relies on Discretionary Access Control (DAC): access permissions are set by users on files they own, namely at the user's discretion. This access control model has proven to be fragile [2].

That is why access decisions have been moved from user control to a Mandatory Access Control (MAC) associated with a protection policy that describes how subjects and objects are allowed to interact. For example, the NSA's Security-Enhanced Linux (SELinux) has been developed as part of the Linux kernel. It implements a strong and highly configurable MAC. Nevertheless these MAC systems can only guarantee simple security properties (as a process cannot access a file). Complex security properties cannot be ensured by this approach. For example, existing MAC systems do not control information flows involving multiple processes and resources.

The literature shows there is no approach to easing the formalization of the required security properties. The authors, to date, consider only specific properties but do not provide a generic method to formalize a large range of security properties related to integrity or confidentiality. Enforcement is also limited to a subset of the required properties. To the best of our knowledge, our proposal is the first one that 1) enables a large set of security properties to be defined and 2) provides a generic method for enforcing all the properties that are supported by our Security Property Language.

13 security templates are presented in this paper that formalize some well know security properties of the literature and propose new ones. We consider that these properties are the most important ones and some of them have been tested on a honeypot. But, new ones can be defined very easily, either by the definition of a new property or by the composition of the proposed template. Moreover, the 13 proposed templates are very flexible. A security administrator can easily reuse them to express multiple concrete security properties. He can define a concrete property with specific values for the arguments of the template.

An extended Linux kernel, called SPLinux, is available to control all the concrete security properties that are requested by the security administrator. This is the first solution to enforce such a large and precise set of security properties. A security administrator uses our SPL compiler to compute two kinds of inputs: 1) the MAC policy available at the target Operating System and 2) the requested concrete properties defined using the SPL language. The compiler produces all the illegal activities that are allowed by the target MAC policy. Then, the Linux kernel uses a userland application that controls all the illegal activities. Thus, a system call fails according to 1) the DAC policy, 2) the MAC policy and 3) the required concrete properties. So, enforcement of the requested security properties is proposed for a target MAC system e.g. SELinux.

When MAC policies are dynamic, the MAC policy states can become infinite. In order to be able to control those states, evolution constraints are required to reduce the possible

[1]This paper is an extended work of the paper presented at SECURWARE 2009 [1].

states. A meta-policy approach has been proposed in previous works [3]. However, the security administrator needs a tool to help him decide if the meta-policy is safe and if the related MAC policy states can be computed efficiently by the SPL compiler. Our verification tool takes two kinds of inputs: the meta-policy that controls the MAC policies and the required concrete security properties. Our verification system provides at least one decision showing an illegal activity. It can also provide, for a given MAC meta-policy, all the possible illegal activities associated with the enforcement of each property. Thus, the security administrator can decide either to 1) remove a security property that cannot be violated or 2) add a better security property that reduces the number of illegal activities or 3) modify the meta-policy in order to minimize the overhead.

A real application is proposed for protecting high interaction honeypots. Our high interaction honeypot is a highly distributed system. A single meta-policy protects our distributed honeypot against corruption of the various nodes. Despite more than two years of experimentation and numerous intrusions, the target Operating System has never been compromised. This large scale experiment have been precisely described in [4]. Even if this experiment is not a formal proof that the meta-policy protects the honeypot against all the possible attacks, we think that a system that has never been compromised while hosting malicious users suggests the efficiency of the proposed protection mechanism.

## II. Related Work

Under Linux, there are at least four security models available to ensure a Mandatory Access Control policy: SELinux, GRSecurity, SMACK and RSBAC. But none of these solutions can ensure a large set of security properties. At best they can ensure one or two limited properties, such as the Bell and LaPadula confidentiality property or the Biba integrity property. Under the BSD family, solutions such as Trusted BSD (available within the following Operating Systems: FreeBSD, OpenBSD, MacOSX, NetBSD) provide more or less the same kind of Mandatory Access Control as SELinux. But, there again they fail to ensure the great majority of requested security properties.

The major limitation is associated with a transitive closure of allowed system calls that enables a security property to be violated. All the existing approaches fail to manage the causal relationships between the system calls correctly, thus authorizing illegal activities.

Several works address how security properties can be enforced within an Operating System. [5] considers only protection policies and enforcement mechanisms, such as those in MAC systems. Thus, the author does not deal with information flows and more generally does not deal with confidentiality or integrity but rather safety. Solutions such as [6], [7] consider noninterference between privileged and unprivileged entities, which is only one specific property. However, they cannot formalize classes of security properties, such as the prevention of flows between privileged entities.

Many solutions deal with the detection of violations of security properties. Solutions such as [8] detect violation of both confidentiality and integrity between privileged and unprivileged entities. Again, those solutions consider only noninterference and cannot enforce other classes of security properties.

[9], [10] detects the flows violating a DAC policy. This is a limited property. Moreover, its practical usage is very limited for DAC systems since a DAC policy is not safe and the related detections include numerous false positives and false negatives.

[11] counts more than 150 publications related to information-flow security. The majority deal with noninterference. The other part aims at enforcing information-flow policies using program analysis techniques. Information flow analysis in a program language is not suited to enforcing protection between several processes that are using the services of an Operating System.

Finally, many works address the conformance of policies. For example, [12] verifies that XACML access control policies really match the Bell and LaPadula, Biba or Chinese Wall models. But, those solutions cannot manage a large range of security properties and do not deal with dynamic policies. [13] considers the verification of SELinux policies. However, it does not provide any administration language to ease the formalization of the required security properties, nor does it manage dynamic MAC policies.

The HiStar Operating System [14] associates each objet or subject with an information flow level. Four different information levels are proposed. For example, a subject with level 1, cannot read an object with level 3. In practice, those levels are very close to capabilities. In HiStar a capacity is a collection of read and write levels that must be consistent with the requested object level. The problem of HiStar is that it is very close to the Bell and Lapadula model and it suffers from the same limitations.

The Flume Operating System [15] proposes a reduced form of MAC policy. They consider two kinds of processes, unconfined and confined. A confined process cannot access the file systems through dedicated system calls. The control of unconfined processes requires a dedicated MAC policy between security contexts. In contrast with SELinux, it is not a fine grain MAC system. Flume is in essence very close to HiStar since it uses capabilities as security contexts. However, Flume does not control information flows.

Asbestos [16] reuses the idea behind HiStar by considering four different levels of information. The protection rules can only express pairwise relationship patterns. Again, information flows involving multiple interactions and processes cannot be controlled easily.

Works related to the enforcement of dynamic policies, such as [17], [18], generally consider how to detect simple conflicts within dynamic policies. For example, they detect if it is safe to remove or add a role or a context, otherwise the considered access control could become invalid, conflicting or unsupported. So, they address conflicting rules but do not

enforce a large set of security properties.

To the best of our knowledge, even recent works such as [16], [15], [14], [19], [20], [21] do not provide any administration language that enables a large range of security properties to be easily formalised. All the Operating Systems described in the literature fail to guarantee the requested security properties. The security of Operating Systems thus remains an open problem. The literature does not address the enforcement of precise and flexible security properties related to integrity and confidentiality. Finally, work on dynamic policies mainly studies how to deal with conflicting protection rules. However, available tools are not able to compute the illegal activities for dynamic MAC policies in order to enforce a large set of security properties. Finally, there has been, as yet, no real experiment. Our paper addresses all the above points.

## III. Abstract Security Language

In order to formalize security properties associated with the activities of the Operating System, let us firstly define the model of the target MAC systems. That model fits the majority of MAC systems. Some of them do not present certain of the modelized functionalities, such as the control of process transitions. In such cases, our formalization is reduced to the functionalities provided by the target MAC system. However, our modelling is adapted for a MAC system providing a fine grain control such as SELinux. For other MAC systems, extensions can be implemented to be able to enforce all the required security properties. Details about how to extend existing Linux systems to enforce the required security properties are available in [22]. In that paper, an abstract Security Property Language is provided to model the required security properties. Hereafter, a concrete Security Property Language uses that abstract language and applications are given for the SELinux systems.

Since the causal dependencies between the system calls and the formalization of the system activities are poorly addressed in the literature, this section defines all the notions of our abstract SPL. In the next section IV, the abstract SPL language enables us to formalize 13 security templates.

### A. System representation

A system is modellised by a set of security contexts performing operations on other security contexts. The security contexts identify the various entities of the system. Let us denote as $SC$ the whole set of security contexts existing in a given system. Two sets of security contexts are considered ($SC = SCs \cup SCo$). $SCo$ is the set of security contexts acting as an object: each $sc_o \in SCo$ characterizes a passive entity (file, socket, ...) on which system calls can be performed. $SCs$ is the set of security contexts acting as a subject: each $sc_s \in SCs$ characterizes an active entity, i.e. processes, that can perform actions, i.e. system calls. For example, let us consider the Apache webserver reading an HTML file, the Apache process is identified as a subject ($apache\_t \in SCs$) and the file is considered as an object ($var\_www\_t \in SCo$).

Let us denote as $IS$ (Interaction Set) the set of all the elementary operations, i.e. system calls, existing in the system (read, write, execute, ...). An interaction $it$ represents a subject $sc_s \in SCs$ executing an operation $eo \in IS$ on a given context $sc_t \in SC$. An interaction is a 3-uple defined by:

$$it = (sc_s \in SCs, sc_t \in SC, eo \in IS)$$

noted:

$$sc_s \xrightarrow{eo} sc_t$$

When an operation is performed, there are two consequences from the security point of view. The interaction can produce:

- an information transfer noted $sc_1 > sc_2$;
- a transition $sc_s \xrightarrow{trans} sc_t$.

An *information transfer* conveys information from context $sc_1$ to $sc_2$ using a write-like operation or a read-like operation. For example, the read interaction $sc_{apache} \xrightarrow{file:read} sc_{var\_www}$ corresponds to the information transfer $sc_{var\_www} > sc_{apache}$ and the write interaction $sc_{apache} \xrightarrow{file:write} sc_{var\_www}$ corresponds to the information transfer $sc_{apache} > sc_{var\_www}$. A precise definition about read and write-like operations is given later in section III-D.

A *transition* enables a process to move from context $sc_s$ to context $sc_t$. When the process is running in the context $sc_t$, it acquires new privileges associated with $sc_t$. For example, the $sc_{init}$ process launches the Apache web server using a forked process that transits using transition $sc_{init} \xrightarrow{process:transition} sc_{apache}$.

### B. Causal dependency

The state of the art related to causality shows that there is no relevant definition of causal dependency to express security properties. The difficulty lies in having a satisfactory estimator of causality between interactions. A new definition of causal dependency is given in this paper between a source and a target context. Two interactions are causally dependant if:

- those interactions share a common security context
- the first interaction occurs before the end of the second interaction
- the first interaction modifies the state of the shared security context
- the second interaction modifies the state of the final security context

The figure 1 gives an example of a causal dependency between two interactions $it_1$ and $it_2$. This example clearly shows that:

1) $sc_2$ is a shared context,
2) $it_1$ finishes before the end of $it_2$,
3) an information flow occurs from $sc_1$ to $sc_2$,
4) an information flow occurs from $sc_2$ to $sc_3$.

*Definition 3.1:* The causal dependency between two interaction $it_1$ and $it_2$, noted $it_1 \nrightarrow it_2$, is defined by

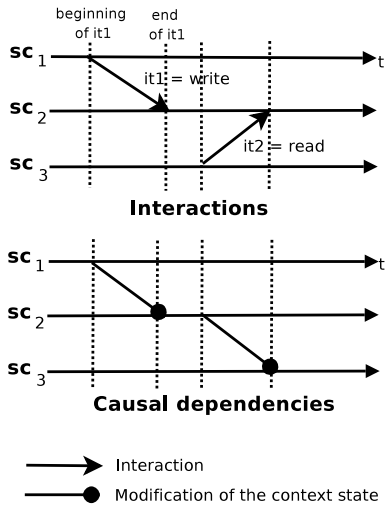


Fig. 1. Read and Write interaction / Causal dependency

$$\begin{cases} sc_2 \in it_1, \\ sc_2 \in it_2, sc_3 \in it_2, \\ date_{sc_2}(begin(it_1)) \leqslant date_{sc_2}(end(it_2)), \\ it_1 \text{ modifies the state of the shared context } sc_2, \\ it_2 \text{ modifies the state of the security context } sc_3. \end{cases}$$

That definition overestimates the probability that the information can flow from the source to the target. However, it is a satisfactory estimator which only takes into account the possible causal relationships. Our abstract SPL efficiently uses that causal dependency to make the formalization of advanced security properties easier.

### C. System activities

In this section, an abstract language is proposed to describe the activities of the different processes. That language uses our definition of causal dependency to model the activities presenting indirect flows. Several operators enable those activities to be combined.

A system activity is defined by a set of interactions. We propose to distinguish four activity classes:

- The *interaction class* is the class where each activity includes a single interaction that can permit direct flows.
- The *sequence class* contains activities composed of sequences of interactions. It enables indirect flows involving several interactions to be modelled.
- The *correlation class* is the class where activities are a combination of interactions and/or sequences.
- The last class of interactions are the remaining activities which are not expressed by our language, i.e. activities that cannot be observed by the target Operating System.

*1) Sequence class:* A sequence of interaction is a transitive closure of causally dependent interactions. It models indirect flows.

*Definition 3.2:* A sequence of $n$ interactions, noted $sc_{source} \Rightarrow sc_{target}$, from a context $sc_{source}$ to $sc_{target}$ is a transitive closure of $n-1$ causal dependencies

$$s.a. \begin{cases} n \geq 2, \\ sc_{source} \in it_1, \\ sc_{target} \in it_n, \\ \forall k = 1..n-1, it_k \twoheadrightarrow it_{k+1} \end{cases}$$

As performed in section III-A we distinguish two kinds of sequences, first, an *information flow* where each interaction is an information transfer, second a *transition sequence* where each interaction of the sequence is a transition. Formally, we obtain:

*Definition 3.3:* An information flow between $sc_{source}$ and $sc_{target}$, noted $sc_{source} \gg sc_{target}$, is a sequence $sc_{source} \Rightarrow sc_{target}$

$$s.a. \begin{cases} sc_{source} \Rightarrow sc_{target} = \{it_1, \ldots, it_n\}, \\ \forall k = 1..n, it_k = sc_k > sc_{k+1} \end{cases}$$

*Definition 3.4:* A transition sequence between $sc_{source}$ and $sc_{target}$, noted $sc_{source} \Rightarrow_{trans} sc_{target}$, is a sequence $sc_{source} \Rightarrow sc_{target}$

$$s.a. \begin{cases} sc_{source} \Rightarrow sc_{target} = \{it_1, \ldots, it_n\}, \\ \forall k = 1..n, it_k = sc_k \xrightarrow{trans} sc_{k+1} \end{cases}$$

Using the example in figure 1, the sequence $\{(sc_1, sc_2, \{file : write\}), (sc_3, sc_2, \{file : read\})\}$ is an information flow $sc_1 \gg sc_3$ composed of two transfers: $sc_1 > sc_2$ and $sc_2 > sc_3$. In the case of two transitions, we obtain the transition sequence $sc_1 \Rightarrow_{trans} sc_3$.

*2) Correlation class:* A correlation is a combination of several sequences and/or interactions. It represents a complex activity of the system. For example, a user can access an Apache information if that user transits to the Apache context and then reads the Apache configuration . This activity is composed of one sequence of transition $sc_{user} \Rightarrow_{trans} sc_{apache}$ and one interaction $sc_{apache\_conf\_t} > sc_{apache}$. To describe the relationships between the elements of the correlation, we define three operators $\circ$, $\wedge$ and $\vee$.

For each sequence $f = sc_1 \xrightarrow{eo_1} \ldots \xrightarrow{eo_{n-1}} sc_n$, noted $sc_1 \Rightarrow sc_n$, let us define a function $F$ which associates the last context $sc_n$ to the first context $sc_1$, that is: $F(sc_1) = sc_n$.

*Definition 3.5:* Let $f = sc_{f_1} \Rightarrow sc_{f_n}$ and $g = sc_{g_1} \Rightarrow sc_{g_m}$ two sequences where $sc_{f_n} = sc_{g_1}$. Let $F$ and $G$ be the corresponding functions.
The composition of two sequences $f$ and $g$ corresponds to a global sequence $sc_{f_1} \Rightarrow (sc_{f_n} = sc_{g_1}) \Rightarrow sc_{g_m}$ that respects the equation: $G \circ F(sc_{f_1}) = G[F(sc_{f_1})] = G(sc_{f_n}) = G(sc_{g_1}) = sc_{g_m}$. We note $g \circ f$ this new sequence $sc_{f_1} \Rightarrow sc_{g_m}$.

*Definition 3.6:* Let $a$ and $b$ two interactions, sequences or correlations, $(a \wedge b)$ is observed if $a$ is observed and $b$ is observed.

*Definition 3.7:* Let $a$ and $b$ two interactions, sequences or correlations, $(a \vee b)$ is observed if $a$ is observed or $b$ is observed.

$$
\begin{array}{rcl}
activity & ::= & [\ description\ "="\ ]\ correlation \\
correlation & ::= & (correlation \wedge correlation) \\
 & & |(correlation \vee correlation) \\
 & & |composition \\
composition & ::= & (composition \circ composition) \\
 & & |terminal \\
terminal & ::= & sequence|interaction \\
interaction & ::= & sc \xrightarrow{eo} sc|sc > sc|sc \xrightarrow{trans} sc \\
sequence & ::= & sc \Rightarrow sc|sc \gg sc|sc \Rightarrow_{trans} sc \\
sc & ::= & "\ security\ context\ " \\
eo & ::= & "\ elementary\ operation\ " \\
description & ::= & "\ name\ of\ activity\ "
\end{array}
$$

Fig. 2. Grammar of activities

We give two particular definitions for the correlation class using the defined operators. The first definition gives a subject access to a special privilege on a given object.

*Definition 3.8:* Let $seq = sc_{source} \Rightarrow sc_{inter}$ be an interaction sequence and $it = sc_{inter} \xrightarrow{eo} sc_{target}$, the composition $(it \circ seq)$ overestimates the possibility of the privilege access $eo$ on $sc_{target}$ by $sc_{source}$. It is noted: $sc_{source} \Rightarrow_{eo} sc_{target}$.

The second definition represents a subject access to the information of a given object:

*Definition 3.9:* Let $seq = sc_{acces} \Rightarrow sc_{inter}$ be a sequence of interactions and $flow = sc_{info} \gg sc_{inter}$ an information flow, the correlation $(seq \wedge flow)$ overestimates the possibility of accessing the information contained in $sc_{info}$ by $sc_{acces}$. It is noted $sc_{acces} \Rrightarrow sc_{info}$.

These two correlations overestimate the possibility of accessing a privilege or an information. We call these two cases an *indirect access* to privilege/information. We define two subcases that improve the estimation: if the sequence of interaction $seq = sc_{acces} \Rightarrow sc_{inter}$ is a sequence of transitions $seq = sc_{acces} \Rrightarrow_{trans} sc_{inter}$ this means that the process has a direct access to the privilege/information. We call this case a *direct access* to privilege/information.

These operators are used to define the grammar of the language that describes all the possible activities of the three classes (interactions, sequences, correlation), presented in figure III-C2. With this grammar we can express complex activities, as shown in the following example: the activity $((it \circ seq_{trans_1}) \wedge (seq_{trans_2} \wedge (flow \vee seq_{int}))$ begins with a transition sequence $seq_{trans_1}$ that permits the interaction $it$; the activity is also composed of a sequence of transitions $seq_{trans_2}$ and a sequence of information transfer $flow$ or a sequence of interactions $seq_{int}$.

*3) Activities overview:* Table I gives an overview of all possible activities modelled in this section. These activities are separated into three classes: interactions, sequences, and correlations. The top part of the Table describes these three classes without any assumption about the nature of each operation. In the bottom part, the activities that characterize information flow, transition and privilege access are summed-up with their associated notations.

### D. Complementary definitions

Additional functions are needed to model the security properties that will be presented in section IV. These functions mainly characterise the operations of $IS$.

*Definition 3.10:* $\texttt{is\_write\_like}:IS \rightarrow \{true, false\}$ is the function that says if an operation modifies an object and can be assimilated to the write operation.

*Definition 3.11:* $\texttt{is\_read\_like}:IS \rightarrow \{true, false\}$ is the function that says if an operation gets information from an object and can be assimilated to the read operation.

*Definition 3.12:* $\texttt{is\_execute\_like}:IS \rightarrow \{true, false\}$ is the function that says if an operation can execute an object.

*Definition 3.13:* $\texttt{is\_add\_like}:IS \rightarrow \{true, false\}$ is the function that says if an operation can add information to an object without being able to read the previous written information in this object.

## IV. Security properties

This section specifies the proposed template of security properties using the activity language defined in the previous section III. These 13 templates include the most known security properties of the literature but include also new security properties in order to show how powerful our language is. This set of templates is not limited. Other templates can be easily defined using the concrete SPL that implements the abstract SPL.

The 13 proposed properties are the ones that we setup on our honeypot to experiment them [4]. The obtained results are briefly described in section VIII. The classical properties of the literature will not be new for the reader: integrity, confidentiality and the well known variants like the Biba integrity or the Bell and LaPadula confidentiality property are examples that enters the proposed model. It shows how the model is able to clearly describe these properties. At the end of the section, the Table II gives an overview of the properties with a name and refers to their formal definition. This is crucial to have such a formal definition of the properties and to be able to show, in the section V how the SPL language will implement these properties.

### A. Integrity

This section proposes four types of integrity security properties from the literature:

- Object integrity, as defined in [23]
- Biba Integrity, which implements the Biba model [24]
- Integrity of subjects, which implements noninterference between processes [8]
- Domain Integrity, which implements a chroot

TABLE I
ACTIVITIES OVERVIEW

| | Interaction | Sequence | Correlation |
|---|---|---|---|
| Neutral | Interaction<br>$sc_1 \xrightarrow{eo} sc_2$ | Interactions sequence<br>$sc_{source} \Rightarrow sc_{target}$ | Access to information<br>$sc_{source} \Rrightarrow sc_{target}$ |
| | | | Privilege access<br>$sc_{source} \Rightarrow_{eo} sc_{target}$ |
| Characterisation | Information transfer<br>$sc_1 > sc_2$ | Information flow<br>$sc_{source} \gg sc_{target}$ | Acces to information by transition<br>$sc_{source} \Rrightarrow_{trans} sc_{target}$ |
| | Transition<br>$sc_1 \xrightarrow{trans} sc_2$ | Sequence of transitions<br>$sc_{source} \Rightarrow_{trans} sc_{target}$ | |
| | | | Privilege access by transition<br>$sc_{source} \Rightarrow_{trans\_eo} sc_{target}$ |

*1) Object integrity:* The integrity of a system object can be altered when a write operation is performed on this object. This object integrity property ensures that no process of the system is able to modify a resource, either directly or by a sequence of interactions. The direct protection of an object is easily obtained with a MAC mechanism. Nevertheless, better integrity protection is required against sequences of operations. For example, if a user uses an exploit on Apache that leads Apache to modify a file of /var/www, this is a sequence of operations that violates the integrity of /var/www.

*Property 4.1:* The integrity of an object $sco_1 \in SC_O$, noted $P_{4.1}(scs_1, sco_1)$, is respected by a subject $scs_1 \in SC_S$ if:

$$\forall eo \in IS \text{ s.a. } scs_1 \xrightarrow{eo} sco_1, \neg is\_write\_like(eo)$$

$$\wedge$$

$$\forall eo \in IS \text{ s.a. } scs_1 \Rightarrow_{eo} sco_1, \neg is\_write\_like(eo)$$

*Property 4.2:* The integrity of a set of objects $SC_{O1} \subset SC_O$, noted $P_{4.2}(SC_{S1}, SC_{O1})$, is respected by a set of subjects $SC_{S1} \subset SC_S$ if:

$$\forall sco_1 \in SC_{O1},\ \forall scs_1 \in SC_{S1},\ P_{4.1}(scs_1, sco_1)$$

*2) Biba Integrity:* The BIBA model [24] defines three rules to guarantee the integrity of the system:

1) To allow $scs_1$ to have read access to an object $sco_1$, its integrity level must be less than or equal to the object integrity level;
2) To allow $scs_1$ to have write access to an object $sco_1$, its integrity level must be greater than or equal to the object integrity level;
3) To allow $scs_1$ to invoke a subject $scs_2$, its integrity level must be greater than or equal to the invoked subject integrity level;

A level of integrity is needed for each entity of the system: let $I : SC \rightarrow \mathbb{N}$ be a function that gives the level of integrity of a security context.

*Property 4.3:* The Biba integrity of an object $sco_1 \in SC_O$, noted $P_{4.3}(scs_1, sco_1)$, is respected by a subject $scs_1 \in SC_S$ if:

$$\forall eo \in IS \text{ s.a.} \begin{cases} scs_1 \xrightarrow{eo} sco_1 \\ is\_read\_like(eo) \end{cases}, I(scs_1) \leqslant I(sco_1)$$

$$\forall eo \in IS \text{ s.a.} \begin{cases} scs_1 \xrightarrow{eo} sco_1 \\ is\_write\_like(eo) \end{cases}, I(scs_1) \geqslant I(sco_1)$$

$$\forall eo \in IS \text{ s.a.} \begin{cases} scs_1 \xrightarrow{eo} sco_1 \\ is\_execute\_like(eo) \end{cases}, I(scs_1) \geqslant I(sco_1)$$

*Property 4.4:* For a system, composed by a set of contexts $SC$, the integrity of this system, noted $P_{4.4}(SC)$, is respected if:

$$\forall scs, sco \in SC_S \times SC_O, P_{4.3}(scs, sco)$$

*3) Integrity of Subjects:* The integrity of subjects corresponds to the non-interference property [25]. The non-interference property uses the notation of commutativity. Two interactions $it_1 = (scs_1, sco_1, eo_1)$ and $it_2 = (scs_2, sco_1, eo_2)$ are commutative if the operation $eo_1$ does not change what the context $scs_2$ sees of $sco_1$. For example, the operations $eo_1 = \{file : write\}$ and $eo_2 = \{file : read\}$ are not commutative, and the operations $eo_1 = \{file : read\}$ and $eo_2 = \{file : read\}$ are commutative. Let $commute : IS \times IS \rightarrow \{true, false\}$ the function that returns the commutativity of two operations using the table defined in [26].

*Property 4.5:* The integrity of a subject $scs_1 \in SC_S$, noted $P_{4.5}(scs_1, scs_2)$, is respected by a subject $scs_2 \in SC_S$ if:

$$\forall sco_1 \in SC_O,\ \forall eo_1, eo_2 \in IS \text{ s.a.} \begin{cases} scs_1 \xrightarrow{eo_1} sco_1 \\ scs_2 \xrightarrow{eo_2} sco_1 \end{cases}, \quad commute(eo_1, eo_2)$$

$$\forall sco_1 \in SC_O,\ \forall eo_1, eo_2 \in IS \text{ s.a.} \begin{cases} scs_1 \Rightarrow_{eo_1} sco_1 \\ scs_2 \Rightarrow_{eo_2} sco_1 \end{cases}, \quad commute(eo_1, eo_2)$$

*Property 4.6:* For a system, composed by a set of subject contexts $SC_S$, the integrity of these subjects, noted $P_{4.6}(SC_S)$, are guaranteed if:

$$\forall scs_1, scs_2 \in SC_S, P_{4.5}(scs_1, scs_2)$$

*4) Domain Integrity:* The property of domain integrity is linked to the notion of *chroot* available in operating systems or software. A set of contexts in a *chroot* are isolated from other outside contexts. Any interaction with a context outside the chroot (the domain) is a violation of the domain property.

*Property 4.7:* Let $chroot_{in} \subset SC$ be a set of contexts representing a domain.
The integrity of the domain $chroot_{in}$, noted $P_{4.7}(chroot_{in})$, is respected if:

$$\begin{array}{l} \forall eo \in IS \\ \forall sc_1, sc_2 \in SC \end{array} \text{, s.a. } sc_1 \xrightarrow{eo} sc_2,$$

$$\begin{cases} sc_1 \in chroot_{in} \Longrightarrow sc_2 \in chroot_{in} \\ \vee \\ sc_2 \in chroot_{in} \Longrightarrow sc_1 \in chroot_{in} \end{cases}$$

### B. Confidentiality

The confidentiality property prevents unwanted accesses. Thus, information flow cannot be operated in order to steal information from an object. Four confidentiality properties are declined in this section:

- Confidentiality of system contexts
- Bell & LaPadula confidentiality
- Bell & LaPadula restrictive confidentiality
- Data access consistency

*1) Confidentiality of system contexts:* This property guarantees that a context $sc_2$ is confidential for the context $sc_1$ if no information flow is possible from $sc_1$ to $sc_2$.

*Property 4.8:* The confidentiality of $sc_2 \in SC$ vs $sc_1 \in SC$, noted $P_{4.8}(sc_1, sc_2)$, is guaranteed if:

$$\begin{cases} \nexists (sc_2 > sc_1) \\ \nexists (sc_2 \gg sc_1) \\ \nexists (sc_1 \Rrightarrow sc_2) \end{cases}$$

*Property 4.9:* For a system, composed of a set of contexts $SC$, the confidentiality of a subset of contexts $SC_1 \subset SC$ vs a subset $SC_2 \subset SC$, noted $P_{4.9}(SC_1, SC_2)$, is guaranteed if:

$$\forall sc_1, sc_2 \in SC_1 \times SC_2,\ P_{4.8}(sc_1, sc_2)$$

*2) Bell & LaPadula confidentiality:* The Bell & LaPadula model uses a level of classification for the subjects and a sensitivity level for objects [27]. Two rules have to be respected in order to guarantee the Bell & LaPadula confidentiality property:

1) if a subject context performs a read operation on an object, its level of classification has to be greater than the sensitivity level of the object;
2) if an information flows from the object $o_2$ to $o_1$, the sensitivity level of $o_2$ must be greater than that of $o_1$.

Let $class : SC \rightarrow \mathbb{N}$ the function that associates a subject with its classification level and for an object its sensitivity level.

*Property 4.10:* The Bell & LaPadula confidentiality property of an object $sco_1 \in SC_O$, noted $P_{4.10}(sco_1, scs)$, is respected by a subject $scs \in SC_S$ if:

$$\exists scs > sco_1,\ class(scs) \geqslant class(sco_1)$$

$$\forall sco_2 \in SC_O \text{ s.a. } sco_2 \gg sco_1,\ class(sco_1) \geqslant class(sco_2)$$

*Property 4.11:* For a system, composed of a set of contexts $SC$, the Bell & LaPadula confidentiality of the system, noted $P_{4.11}(SC)$, is guaranteed if:

$$\forall sco, scs \in SC_O \times SC_S,\ P_{4.10}(sco, scs)$$

*3) Bell & LaPadula restrictive confidentiality:* In contrast with the previous property, the Bell & LaPadula restrictive confidentiality provides a finer grain protection. Three rules are needed to express this property:

1) if a subject context reads an object, its level of classification has to be greater than the sensitivity level of the object;
2) if a subject context adds information to an object (for example, appending text with no possible read and modification of the existing data on a text file), its level of classification has to be less than the sensitivity level of the object;
3) if a subject context modifies an object (for example, read and write classification on a text file), its level of classification has to be equal to the sensitivity level of the object;

*Property 4.12:* The Bell & LaPadula restrictive confidentiality property of an object $sco \in SC_O$, noted $P_{4.12}(sco, scs)$, is respected by a subject $scs \in SC_S$ if:

$$\exists scs > sco_1,\ class(scs) \geqslant class(sco)$$

$$\forall eo \in IS \text{ s.a. } \begin{cases} scs \xrightarrow{eo} sco \\ is\_add\_like(eo) \end{cases}, class(scs) \leqslant class(sco)$$

$$\exists sco_1 > scs,\ class(scs) = class(sco)$$

*Property 4.13:* For a system, composed of a set of contexts $SC$, the Bell & LaPadula restrictive confidentiality of the system, noted $P_{4.13}(SC)$, is guaranteed if:

$$\forall sco, scs \in SC_O \times SC_S,\ P_{4.12}(sco, scs)$$

*4) Data access consistency:* This property aims to guarantee that the access to the data is consistent between direct access (one interaction) and a sequence of interactions that leads to data access. In contrast with data access consistency for DAC systems [10], our property provides a better approach since a MAC policy can be safe. In case of MAC systems, if a context cannot read a file directly, there is no reason for it to be able to read the information using a transition to another context.

*Property 4.14:* The data access consistency property of two contexts $sc_1, sc_2 \in SC^2$, noted $P_{4.14}(sc_1, sc_2)$, is respected if:

$$sc_1 \Rrightarrow sc_2 \text{ implies } sc_2 > sc_1$$

*Property 4.15:* For a system, composed of a set of contexts $SC$, the data access consistency property, noted $P_{4.15}(SC)$, is guaranteed if:

$$\forall sc_1, sc_2 \in SC, P_{4.14}(sc_1, sc_2)$$

### C. Privilege abuse

This class of security property is designed to prevent a malicious use of the privileges available on a system. We distinguish five possible types of privilege abuse:

- The separation of duties
- No context transition
- Trusted path execution
- Policy abuse
- Meta-policy abuse

*1) The Separation of Duties:* The property of the separation of duties is defined in [28] by the idea that an object created by one person cannot be executed by that same person. A more general definition is given in [29]: several operations performed on a same object must be done so by different users. This property must take into account that:

- a subject can perform operations directly on the subject using an interaction: in this case we call the property the separation of direct duties;
- a subject can use a sequence of operations to perform the operation on the object: in this case we call the property the separation of extended duties.

Let us give a more general definition of the separation of duties associated with two sets of operations: $OP_1$ and $OP_2$. Then, this general definition is illustrated with the definition given in [28] that concerns the special write operation.

*Property 4.16:* Let $OP_1, OP_2 \subset IS^2$ two sets of elementary operations. A system composed of contexts $SC = SC_S \cup SC_O$ respects the separation of duties of $OP_1$ and $OP_2$ if:

$$\left.\begin{array}{l}\forall sc_s \in SC_S \\ \forall sc_o \in SC_O \\ \forall eo_1, eo_2 \in IS\end{array}\right. \text{ s.a. } \left\{\begin{array}{l} it_1 = sc_s \xrightarrow{eo_1} sc_o \\ it_2 = sc_s \xrightarrow{eo_2} sc_o \\ eo_1 \in OP_1 \\ (it_2 \circ it_1)\end{array}\right. , eo_2 \notin OP_2$$
$$\wedge$$
$$\left\{\begin{array}{l} it_3 = sc_s \Rightarrow_{eo_1} sc_o \\ it_4 = sc_s \Rightarrow_{eo_2} sc_o \\ eo_1 \in OP_1 \\ (it_4 \circ it_3)\end{array}\right. , eo_2 \notin OP_2$$

The definition of [28], called the separation of execute/write duties property can be implemented with $OP_2 = \{execute\}$ and $OP_1 = \{write\}$:

*Property 4.17:* The separation of execute/write duties property on an object $sco \in SC_O$ for a subject $scs \in SC_S$, noted $P_{4.17}(scs, sco)$, is respected if:

$$\forall eo_1, eo_2 \in IS \text{ s.a. } \left\{\begin{array}{l} it_1 = scs \xrightarrow{eo_1} sco \\ it_2 = scs \xrightarrow{eo_2} sco \\ is\_write\_like(eo_1) \\ (it_2 \circ it_1)\end{array}\right. ,$$
$$\neg is\_execute\_like(eo_2)$$

*Property 4.18:* The separation of execute/write duties property for a subject $scs \in SC_S$ and all possible related objects of the system, noted $P_{4.18}(scs)$, is respected if:

$$\forall sc_o \in SCO,\ P_{4.17}(scs, sco)$$

*Property 4.19:* The separation of extended execute/write duties property on an object $sco \in SC_O$ for a subject $scs \in SC_S$, noted $P_{4.19}(scs, sco)$, is respected if:

$$\forall eo_1, eo_2 \in IS \text{ s.a. } \left\{\begin{array}{l} it_1 = scs \Rightarrow_{eo_1} sco \\ it_2 = scs \Rightarrow_{eo_2} sco \\ is\_write\_like(eo_1) \\ (it_2 \circ it_1)\end{array}\right. ,$$
$$\neg is\_execute\_like(eo_2)$$

*Property 4.20:* For a system, composed of a set of contexts $SC$, the separation of execute/write duties property of the system, noted $P_{4.20}(SC)$, is guaranteed if:

$$\forall scs, sco \in SC_S \times SC_O,\ P_{4.17}(scs, sco) \wedge P_{4.19}(scs, sco)$$

*2) No Context transition:* Let us define a new property called "No context transition" that guarantees that a process will not be able to transit to another context. This can be considered as an integrity property for the process running in this context; it can also be seen as a sort of confidentiality for this process that cannot change its context and bring information into another context.

*Property 4.21:* The no context transition of a subject $scs_1$ with $scs_2$, noted $P_{4.21}(scs_1, scs_2)$, is guaranteed if:

$$\left\{\begin{array}{l} \neg scs_1 \xrightarrow{trans} scs_2 \\ \neg scs_1 \Rightarrow_{trans} scs_2\end{array}\right.$$

A more general version of this property protects one security context that cannot transit to any other context. For example, this property can be applied to the Apache web server that will not be allowed to transit to any other context in case of an attack trying to exploit a vulnerability.

*Property 4.22:* The no context transition of a subject $scs_1$, noted $P_{4.22}(scs_1)$, is guaranteed if:

$$\forall scs_2 \in SC_S \text{ s.a. } P_{4.21}(scs_1, scs_2)$$

*3) Trusted path execution:* This notion of trust refers to the administrators that trust the executables they have installed but does not trust the executables installed by other users. To use this property, a set of contexts noted TPE is built by the administrator to group all the software that he wants to authorize on the system. Any file that can be executed outside of this set breaks the trusted path execution property.

*Property 4.23:* Let $TPE \subset SC$ the set of the trusted contexts. The system guarantees the trusted path execution property, noted $P_{4.23}(TPE)$, if:

$$\forall sc_1, sc_2 \in SC, \forall eo \in IS \text{ s.a } \left\{\begin{array}{l} sc_1 \xrightarrow{eo} sc_2 \\ is\_execute\_like(eo)\end{array}\right. ,$$
$$sc_2 \in TPE$$

*4) Policy abuse:* A MAC policy can be deployed on a system. This policy, noted $\mathcal{POL}$, defines all the allowed interactions on the system. Any operation not included in this policy is a violation attempt.

*Property 4.24:* Let $\mathcal{POL} = \{it_1, \ldots, it_n\}$ be a MAC policy. A system guarantees the respect of the policy abuse property, noted $P_{4.24}(SC, \mathcal{POL})$, if:

$$\forall sc_1, sc_2 \in SC, \forall eo \in IS \text{ s.a. } it = sc_1 \xrightarrow{eo} sc_2,$$

$$it \in \mathcal{POL}$$

*5) Meta-policy abuse:* This property guarantees that a meta-policy, defined in section VI, is respected. A meta-policy constrains a policy by evolution constraints. The policy of the system is then dynamic but respects the meta-policy. An interaction is a violation of the meta-policy if no policy that respects the meta-policy contains this interaction.

*Property 4.25:* Let $\mathcal{MP}$ be a meta-policy. A MAC policy $\mathcal{POL}$ that respects $\mathcal{MP}$ is noted $\mathcal{POL} \in \mathcal{MP}$. A system guarantees the respect of the meta-policy abuse property, noted $P_{4.25}(SC, \mathcal{MP})$, if:

$$\begin{matrix}\forall sc_1, sc_2 \in SC \\ \forall eo \in IS\end{matrix} \text{, s.a. } it = sc_1 \xrightarrow{eo} sc_2,$$

$$\exists \mathcal{POL} \in \mathcal{MP} \text{ s.a. } it \in \mathcal{POL}$$

In practice, this property is hard to check as the control of dynamic MAC policies is complex and can require a high overhead. Section VII covers this problem.

### D. Security properties sum-up

Table II gives an overview of the described security properties. Each property has a precise name, a reference number in the model, a name in the concrete SPL implementation and a short explanation about the property. With these 13 security properties, 13 protection templates will be presented in the next section that gives a real implementation using the concrete SPL language. Nevertheless, it is important to remember that the SPL language can handle more security properties and that only the 13 most important properties are given in this paper.

## V. Concrete Security Property Language

Our abstract SPL language needs a concrete syntax that can be compiled as a protection language. Instead of giving the complete concrete syntax, which is pointless since this implements the previous abstract language, this section describes how each template is expressed using our concrete SPL.

Our concrete SPL enables us to define security functions, i.e. templates. Calling those functions corresponds to the instantiation of the protection template to the target Operating Systems. Each template is very powerful. It enables multiple protection properties to be easily expressed using only specific values for the arguments. Moreover, each concrete security property makes it possible to compute the whole set of illegal activities in contradiction with the requested security properties, as presented in section VII.

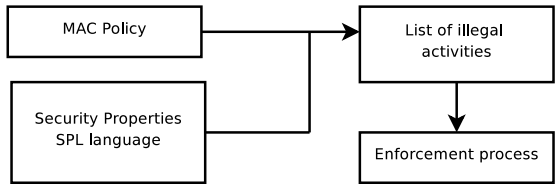


Fig. 3. Enforcement of Security Properties

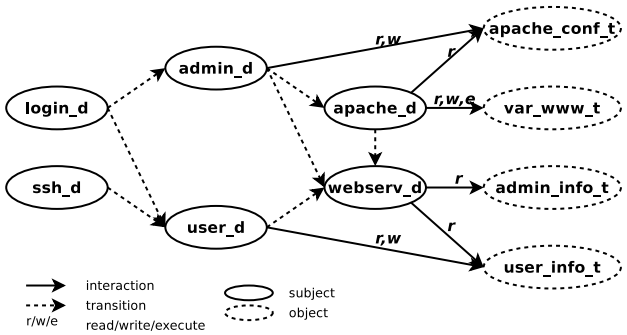


Fig. 4. Example policy for the web server Apache

Those templates are written using a concrete language called SPL, for Security Properties Language, that we have created specifically to be able to quickly write new security properties. A compilation of our concrete SPL language produces all illegal activities i.e. violating one of the security properties [22]. An enforcement process uses those illegal activities and communicates with the SPLinux kernel in order to guarantee the requested properties as represented in figure 3. See [22] to have details about the enforcement architecture that complements the classical DAC and the SELinux protections.

Through the compilation of the function calls, all the violations of the security properties found in the MAC policy are added to the list of illegal activities. These activities can be used to:

- correct the MAC policy if possible;
- to prevent or detect these violations if they occur.

Hereafter, two types of examples are given for each property:

- a simplified example associated with a web server policy, presented on figure 5;
- a complete MAC policy associated with a SELinux host. The whole policy is impossible to represent in this paper as the number of contexts approaches 2,000 and there are more than 100,000 interactions.

### A. Apache policy description

The policy presented in figure 4 is a simplified version of the policy that is used for Apache and for the logging of the users in the system. The users can log in the system using a local console under the *login_d* context or via SSH under the *ssh_d* context. Then, a user can transit into the *user_d* context or to the *administrator_d* context in order to run commands from the shell. Note that a SSH user is not allowed to become an administrator in this policy.

TABLE II
SECURITY PROPERTIES SUM-UP

| Property | Specificity | Reference | Rule | Goal |
|---|---|---|---|---|
| Integrity | Objects Integrity | $P_{4.2}(SC_{S1}, SC_{O1})$ | `integrity` | A set of contexts must not modify another set of contexts |
| | Biba Integrity | $P_{4.4}(SC)$ | `int_biba` | Biba security model applied to a set of contexts |
| | Subjects Integrity | $P_{4.6}(SC_S)$ | `int_subject` | Non-interference property applied on a set of subjects (process) |
| | Domain Integrity | $P_{4.7}(chroot_{in})$ | `int_domain` | Chroot for a set of contexts (virtual chroot) |
| Confi-dentiality | Confidentiality | $P_{4.9}(SC_1, SC_2)$ | `confidentiality` | A set of contexts must not obtain information from another set |
| | Bell&Lapadula | $P_{4.11}(SC)$ | `conf_blp` | Bell & LaPadulla security model applied to a set of contexts |
| | Bell&Lapadula restrictive | $P_{4.13}(SC)$ | `conf_blpr` | Bell & LaPadulla restrictive security model applied to a set of contexts |
| | Data Access Consistency | $P_{4.15}(SC)$ | `conf_data` | A context must not obtain information that it cannot obtain directly |
| Privilege Abuse | Duties Separation | $P_{4.20}(SC)$ | `duties_separation` | A context must not modify and then execute another context |
| | No Context Transition | $P_{4.22}(scs_1)$ | `no_transition` | A context must not access another context by transition |
| | Trusted Path Execution | $P_{4.23}(TPE)$ | `tpe` | All trusted executables contexts must grouped in a specific set |
| | Policy Abuse | $P_{4.24}(SC, \mathcal{POL})$ | `conformity` | The violation attempts of the local policy are forbidden |
| | Meta-Policy Abuse | $P_{4.25}(SC, \mathcal{MP})$ | `meta-conformity` | The possible violation attempts of the meta-policy are forbidden |

The administrator can setup the Apache configuration (read and write permissions on *apache_conf_t*) and can transit to the context *apache_d* in order to launch the service. Then, Apache can read its configuration and use the files under the *var_www_t* context.

A web service context *webserv_d* enables remote users to get information from their account. From that web service, a user authenticates and retrieves personal information (but he cannot modify it) located in the *user_info_t* context. Apache is able to transit in the special context *webserv_d* to obtain the permission to read the information of the user or of the administrator. Of course, a user logged in the system with a shell can also read or write his personal information typed with the *user_info_t* context.

The administrator is also allowed to transit to the *webserv_d* context, because he needs to manage the web service. The administrator is not supposed to run processes in this context that would read the personal information of the users (which represents a violation of confidentiality).

### B. Integrity

The integrity property $P_{4.1}(sc1, sc2)$ corresponds to the SPL function of listing 1. This function takes two arguments: a set of subject contexts and a set of object contexts. For each write operation $eo$ between a subject context $sc1$ and an object context $sc2$, the function checks that such an interaction does not exist in the policy. The same check is done considering a write privilege access (c.f. definition 3.8).

Listing 1. Integrity check function of $P_{4.1}(sc1, sc2)$

```
1  define integrity( $sc1 IN SCS, $sc2 IN SCO ) [
2          Foreach $eo IN is_write_like(IS)
3                  SA { $sc1−> { $eo } $sc2 } ,
4                          { not(exist()) };
5          Foreach $eo IN is_write_like(IS)
6                  SA { $sc1 => { $eo } $sc2 } ,
7                          { not(exist()) };
8  ];
```

*1) Apache example:* Using the policy of figure 4, the administrator can express that he wants to guarantee that any user using SSH will not be able to modify the Apache configuration (as only the local root is supposed to do so). Moreover, the administrator is not supposed to modify the personal information of the user (*user_info_t*).

Listing 2. Integrity security properties for Apache

```
1  integrity( $sc1:="ssh_d", $sc2:="apache_conf_t" );
2  integrity( $sc1:="admin_d", $sc2:="user_info_t" );
```

*2) Operating system example:* For a whole operating system, the first rule of listing 3 guarantees that no user is able to modify the binaries of the system, the second rule guarantees that no user can modify the configuration files of the system in */etc*. That integrity function prevents any ordinary user from modifying those files. This kind of guarantee cannot be obtained by means of a classical integrity checker like AIDE, TRIPWIRE. Nevertheless, the root user, logged locally on the host is able to modify these files.

Listing 3. Integrity security properties example

```
1  integrity( $sc1:="user_u:user_r:user.∗_t", $sc2:=".∗:.∗:.∗_exec_t" );
2  integrity( $sc1:="user_u:user_r:user.∗_t", $sc2:=".∗:.∗:.∗etc_t" );
```

### C. Domain integrity

The integrity property $P_{4.7}(chroot_{in})$ corresponds to the listing function 4. This property allows it to virtually *chroot* a set of contexts.

Listing 4. Domain integrity check function of $P_{4.7}(chroot_{in})$

```
1  define int_domain( $CHROOT IN SC ) [
2          Foreach $eo IN IS, Foreach $sc1 IN $CHROOT, Foreach $sc2
                   IN SC
3                  SA { $sc1 −> { $eo } $sc2 } ,
4                          { $sc2 IN $CHROOT };
5          Foreach $eo IN IS, Foreach $sc1 IN SCS, Foreach $sc2 IN
                   $CHROOT
6                  SA { $sc1 −> { $eo } $sc2 } ,
7                          { $sc1 IN $CHROOT };
8  ];
```

*1) Apache example:* This property prevents an exploit against Apache enabling a user to gain root privileges, for example.

Listing 5. Domain integrity for apache

```
1  int_domain( $CHROOT:="apache.∗", "webserv_d", ".∗info_t");
```

This property cannot be enforced at start-up. During the boot sequence, the property is not available. It is enforced after the starting of the Apache process. Moreover, it prevents an attacker from restarting the web server. The principle of guaranteeing online properties is described in [22].

*2) Operating system example:* A user can be *chrooted* in his home directory. Again, this system instantiates this property after the boot sequence and the login of the user (otherwise, the user cannot log into the system). Thus, an exploit in the user domain cannot leave that domain.

Listing 6. Domain integrity security property example

```
1  int_domain( $CHROOT:="user_u:.*:.*" );
```

### D. Confidentiality

The confidentiality property $P_{4.8}(sc_1, sc_2)$ corresponds to the function of listing 7. This property checks that no information transfer or flow exists between two contexts.

Listing 7. Confidentiality check function of $P_{4.8}(sc_1, sc_2)$

```
1  define confidentiality( $sc1 IN SCS, $sc2 IN SCO ) [
2          SA { $sc2 > $sc1 },
3                  { not(exist()) };
4          SA { $sc2 >> $sc1 },
5                  { not(exist()) };
6  ];
```

*1) Apache example:* For Apache, the configuration files of the web server must be confidential for the users entering the system via SSH. Moreover, the administrator must not access the personal information of the users in the context *user_info_t*. In contrast, the users must not access the personal data of the administrator.

Listing 8. Confidentiality for Apache

```
1  confidentiality( $sc1 := "ssh_d", $sc2 := "apache_conf_t" );
2  confidentiality( $sc1 := "admin_d", $sc2 := "user_info_t" );
3  confidentiality( $sc1 := "user_d", $sc2 := "admin_info_t" );
```

*2) Operating system example:* The goal is to protect the files */etc/shadow* which contain the hashed user passwords, the special file */dev/mem* that gives access to the memory space of the processes. For software such as Firefox, the confidentiality property prevents any leak of information of the cookies and cache files. For example, if the user launches a malware, the malware will be unable to read the Firefox cookies or cache.

Listing 9. Confidentiality security properties example

```
1  confidentiality( $sc1:="user_u:user_r:user.*_t", $sc2:=system_u:
       object_r:shadow_t );
2  confidentiality( $sc1:="user_u:user_r:user.*_t", $sc2:=system_u:
       object_r:memory_device_t );
3
4  confidentiality( $sc1:=user_u:user_r:user_t, $sc2:=user_u:object_r:
       user_mozilla_cookie_t );
5  confidentiality( $sc1:=user_u:user_r:user_t, $sc2:=user_u:object_r:
       user_mozilla_cache_t );
```

### E. Data access consistency

The data consistency property $P_{4.14}(sc_1, sc_2)$ corresponds to the function of listing 10. This property allows indirect access only if direct access is allowed.

Listing 10. Data access consistency check function $P_{4.14}(sc_1, sc_2)$

```
1  define conf_data($sc1 IN SC, $sc2 IN SC) [
2          SA { $sc1 >>> $sc2 },
3                  { exist[$sc2 > $sc1] };
4  ];
```

*1) Apache example:* For Apache, the property of listing 11 controls the access to personal information files for all the users. The user will be able to access to his personal information (*user_info_t*) using the web service, because he is allowed to read it directly. In contrast, he cannot access the personal information of the administrator using the web service (or using a vulnerability of the web service) because the user has no direct permission on the *admin_info_t* context.

Listing 11. Data access for apache

```
1  conf_data( $sc1 := "user_d", $sc2 := ".*_info_t" );
```

*2) Operating system example:* The property of listing 12 protects the system from the users. For example, if a user becomes root, he will not be able to read or write the */etc/shadow* file, because he has no direct permission on it.

Listing 12. Data access consistency security property example

```
1  conf_data($sc1:="user_u:user_r:user.*_t" , $sc2 := "system_u:.*:.*" );
```

### F. Duties separation

The separation of duties property $P_{4.18}(scs)$ corresponds to the function of listing 13. This function controls the modification of a file and prevents its execution. A special security property has been written for bash scripts and is explained in the example of section V-F2.

Listing 13. Duties separation check function of $P_{4.18}(scs)$ and a special security property for interpreted scripts

```
1  define duties_separation( $sc1 IN SC ) [
2          Foreach $eo1 IN is_write_like(IS), Foreach $eo2 IN
               is_execute_like(IS), Foreach $sc2 IN SCO
3                  SA { ($sc1 −> { $eo2 } $sc2 o $sc1 −> { $eo1 }
                       $sc2) } ,
4                    { not( exist() ) };
5  ];
6  define dutiesseparationbash( $sc1 IN SC ) [
7          Foreach $eo1 IN is_write_like(IS), Foreach $eo2 IN
               is_execute_like(IS), Foreach $eo3 IN is_read_like(IS),
8          Foreach $sc2 IN SCO, Foreach $sc3 IN SC,
9          Foreach $a1 IN ACT, Foreach $a2 IN ACT
10                 SA { ( [ $a2 := $sc1 −> { $eo3 } $sc2 ] o ( [ $a1 :=
                       $sc1 −> { $eo2 } $sc3 ] o $sc1 −> { $eo1 }
                       $sc2) ) } ,
11                   { INHERIT($a2 , $a1) };
12 ];
```

*1) Apache example:* The property of listing 14 prevents the execution of a file written by Apache in */var/www*. It avoids the exploit of a vulnerability against Apache that force Apache to write a script and then to execute it.

Listing 14. Duties separation for Apache

```
1  duties_separation( $sc1 := "apache_d" );
```

*2) Operating system example:* For a system, the property of listing 15 restricts the rights of a user: if the user downloads a program, he will not be able to execute it. The second rule of listing 15 prevents the writing of a script and its execution. This is a special rule because the binary file that is executed is not the script but the bash shell itself that

reads the modified script. The function *dutiesseparationbash* of listing 4.18 checks that the created bash process is inherited from the user process (to identify the user that launched the script) and prevents the reading of the script by the bash process.

Listing 15. Duties separation security property example

```
1  duties_separation( $sc1:="user_u:user_r:user.*_t" );
2  dutiesseparationbash( $sc1:="user_u:user_r:user.*_t" );
```

### G. Trusted path execution

The trusted path execution property $P_{4.23}(TPE)$ corresponds to the function of listing 16. The first function defines the set of binaries that are executables. The second function adds a set of source context that are allowed to execute the binaries and to read the libraries of the set $TPE$ (because an executed binary reads libraries).

Listing 16. Trusted path execution check function of $P_{4.23}(TPE)$

```
1  define tpe( $TPE IN SC ) [
2          Foreach $sc1 IN SCS, Foreach $sc2 IN SC, Foreach $eo IN
               is_execute_like(IS)
3            SA { $sc1 –> { $eo } $sc2 } ,
4                  { ( $sc2 IN $TPE ) };
5  ];
6
7  define tpeuser( $sc1 IN SCS, $TPE IN SC ) [
8          Foreach $sc2 IN SC, Foreach $eo1 IN is_execute_like(IS),
               Foreach $eo2 IN is_read_like(IS)
9            SA { ( $sc1 –> { $eo1 } $sc2 OR $sc1 –> { $eo2 }
                   $sc2 ) } ,
10                 { ( $sc2 IN $TPE ) };
11 ];
```

*1) Apache example:* The property of listing 17 allows only the execution of the authentication programs, the web server and web service. Even if the MAC policy or the UNIX rights (uog+x) authorize the execution permission for programs in */var/www*, this property will prevent the execution of these programs. If an attacker exploits a vulnerability, he can force Apache to execute malware (downloaded file). The trusted path property prevents the execution of that malware.

Listing 17. Trusted path execution for Apache

```
1  tpe( $sc1 := {"login_d", "ssh_d", "apache_d", "webserv_d"} );
```

*2) Operating system example:* The first $tpe$ rule of listing 18 defines the classical executables of the system that all users and daemons can use. The second rule $tpeuser$ is more restrictive than the first one: it allows only the users to execute libraries, Firefox, Claws Mail and Open Office. The last rule is a special rule for Open Office that needs the execution of a shell to be able to launch itself.

Listing 18. Trusted path execution security property example

```
1  tpe( $TPE:= { ".*:.*:.*bin_t", ".*:.*:.*exec_t", ".*:.*:.*lib_t", "system_u:
       object_r:ld_so_t" } );
2  tpeuser( $sc1:=user_u:user_r:user_t, $TPE:= { ".*:.*:.*lib_t", "
       system_u:object_r:mozilla_exec_t", "system_u:object_r:
       clawsmail_exec_t", "system_u:object_r:ooffice_exec_t", "
       system_u:object_r:ld_so_t" } );
3  tpeuser( $sc1:=user_u:user_r:user_ooffice_t, $TPE:= { "user_u:user_r:
       user_ooffice_t", "system_u:object_r:bin_t","system_u:object_r:
       ld_so_t","system_u:object_r:lib_t","system_u:object_r:
       ooffice_exec_t","system_u:object_r:ooffice_lib_t","system_u:
       object_r:shell_exec_t" } );
```

### H. No context transition

The no context transition property $P_{4.22}(scs_1)$ corresponds to the function of listing 19. This function checks that no transition is possible for the context given in the parameters.

Listing 19. No context transition check function

```
1  define no_transition( $sc1 IN $SCNoTransition ) [
2          Foreach $sc2 IN SCS
3                  SA { $sc1 __> $sc2 },
4                          { not(exist()) };
5  ];
```

*1) Apache example:* The property of listing 20 guarantees that the web service cannot transit to another context. For example, an attacker that exploits a vulnerability can try to force the web service to transit to the *admin_d* or *user_d* contexts to obtain the privileges associated with these contexts.

Listing 20. No context transition for Apache

```
1  no_transition( $sc1 := "webserv_d" );
```

*2) Operating system example:* The first rule of listing 21 prevents an attacker from exploiting a vulnerability on Apache, Php or Mysql services to get more privileges on the system (for example, the root privileges). The second rule prevents the user from transitting to another context, such as the root or Apache domain.

Listing 21. Confidentiality property example

```
1  no_transition( $SCNoTransition:= { "system_u:system_r:httpd_t","
       system_u:system_r:httpd_php_t","system_u:system_r:mysql_t" }
       );
2  no_transition( $SCNoTransition:= { "user_u:user_r:user_t" } );
```

## VI. Summary of our meta-policy approach

Before moving on to the verification section of dynamic policies in section VII, this section recalls the basics of our meta-policy [3] approach. The meta-policy provides a means to reduce the space of possible MAC policies, adding constraints on the allowed policies. This way, it becomes possible for the security properties for dynamic policies to be verified, as presented in section VII. This section gives a short overview of our meta-policy model and an example of use.

### A. Meta-policy model

This section gives a short overview of the concept of meta-policy. Further details are given in [3]. It allows dynamic policies i.e. the evolution of multiple local MAC policies whose states are constrained by modification rules.

*1) Initial Policy:* Our meta-policy contains an initial policy i.e. a set of Interaction Vectors. The generic rule for enabling a set of Interaction Vectors is as follows:

```
1  enableIV(pattern_S, pattern_O, pattern_oper)
```

$pattern_S$ is a regular expression that designates the subject security contexts that are considered in this rule. $pattern_O$ designates the object security contexts and $pattern_{oper}$ defines the different operations to be allowed between these contexts.

*2) Modification rules:* Each local policy can evolve according to modification rules. A modification rule includes different elements: the security context allowed to request the modification and the Interaction Vectors. The considered modifications are: addition, removal or change.

1 enableAddIV($sc_{requester}$, ($pattern_S$, $pattern_O$,$pattern_{oper}$))
2 enableModIV($sc_{requester}$, ($pattern_S$, $pattern_O$,$pattern_{oper}$))
3 enableDelIV($sc_{requester}$, ($pattern_S$, $pattern_O$,$pattern_{oper}$))

The security context $sc_{requester} \in SCs$ is the one that can request the modification for $iv = (pattern_S, pattern_O, pattern_{oper})$. For example, $enableAddIV(sc_{admin}, (apache\_(.*), var\_www\_(.*), \{file : (.*)\}))$ can be used to authorize a local administrator $sc_{admin}$ to add the interactions permitting a HTTP process to access any file located in the directory */var/www* with the required privileges. A second set of rules is used to control security context modifications:

1 enableAddSC($sc_{requester}$, $pattern_{SC}$)
2 enableDelSC($sc_{requester}$, $pattern_{SC}$)

For example, $enableAddSC(sc_{admin}, apache\_(.*))$ enables a local administrator to add required Apache server security contexts. Note that similar rules are defined in order to label an "object" with a security context or to modify this labelling.

### B. Example

Listing 22. Meta-policy example

```
1  enableIV( login_d, admin_d, transition )
2  enableIV( login_d, user_d, transition )
3  enableIV( ssh_d, user_d, transition )
4  enableIV( user_d, webserv_d, transition )
5  enableIV( admin_d, webserv_d, transition )
6  enableIV( admin_d, apache_d, transition )
7  enableIV( apache_d, webserv_d, transition )
8  enableIV( admin_d, apache_conf_t, { read, write } )
9  enableIV( apache_d, apache_conf_t, { read } )
10 enableIV( apache_d, var_www_t, { read, write, execute } )
11 enableIV( webserv_d, .*_info_t, { read } )
12 enableIV( user_d, user_info_t, { read, write } )
13
14 enableAddSC(webserv_d, .*_info_t)
15 enableAddIV( webserv_d, (webserv_d, .*_info_t, { read } ))
16 enableDelIV( webserv_d, (webserv_d, .*_info_t, { read } ))
```

The listing 22 contains an example of meta-policy. In this example, a user can open a session with *ssh* or local login, but the administrator can only authenticate locally. Both of them can modify objects containing personal information and execute a web-service allowed to access this information. Moreover, the administrator can execute an Apache web-server and modify its configuration. Apache can only read this configuration, but it is allowed to read, write or execute temporary objects. Finally, Apache can also execute the web-service.

### C. Flow graphs between the security contexts

Starting from a MAC policy e.g. the initial policy of our meta-policy, we have built two different graphs that will be used to check the security properties (as described in the next section). These graphs are the following:

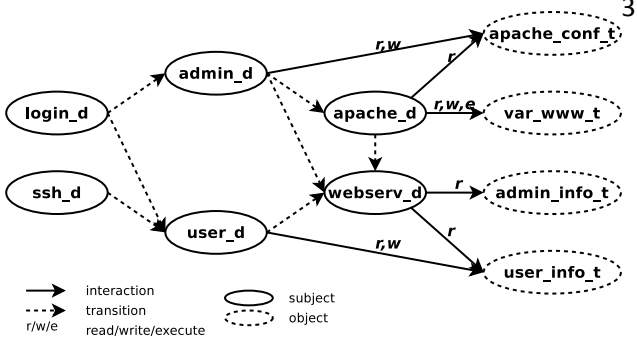


Fig. 5. Interaction graph

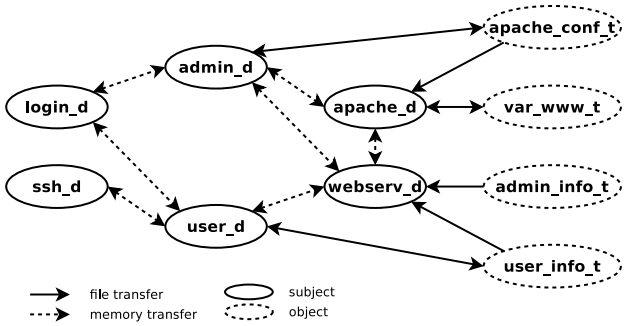


Fig. 6. Information flow graph

*1) Interaction Graph:* The interaction graph is a representation of all legal interactions allowed by the local policy. An interaction between two security contexts $sc_1$ and $sc_2$ is represented in $G$ by an arc , noted $sc_1 \rightarrow_{eo} sc_2$, valuated by the set of authorized operations eo. Figure 5 presents an interaction graph for the previous meta-policy of listing 22. This graph is composed of six subjects and four objects.

*2) Flow Graphs:* Several flow graphs are computed starting from the previous interaction graph. The direct flows are included within the transitive flows i.e. an indirect path between two contexts. In order to simplify the explanations and focus on the enforcement of dynamic policies, limited details are given about those different graphs. Let us give an explanation for the information flow graph. This graph represents all the allowed transfer of information.

A transfer of information between two security contexts $sc_1$ and $sc_2$ is in fact a *read* (*write*) operation from $sc_2$ to $sc_1$ (from $sc_1$ to $sc_2$). Note that a *transition* from $sc_1$ to $sc_2$ allows a transfer of information in both directions. The information graph is a conversion of the interaction graph: the read arcs are flipped, an arc $sc_{s2} \rightarrow sc_{s1}$ is added for each transition arc $sc_{s1} \rightarrow sc_{s2}$.

Figure 6 represents the information flow graph corresponding to the previous example. Another flow graph is computed that includes all the allowed flows of control. A flow of control is a causal sequence including several reading or writing-like operations that provides a path for executing or writing an object.

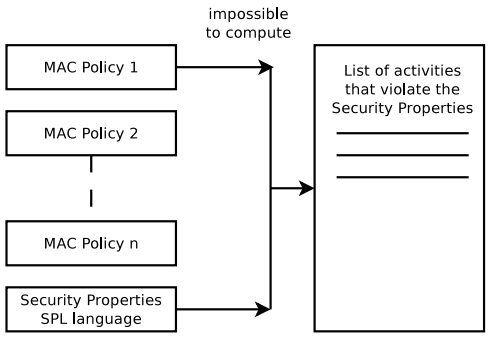


Fig. 7. Impossibility of computing the illegal activities for dynamic policies without a tool based on the meta-policy approach

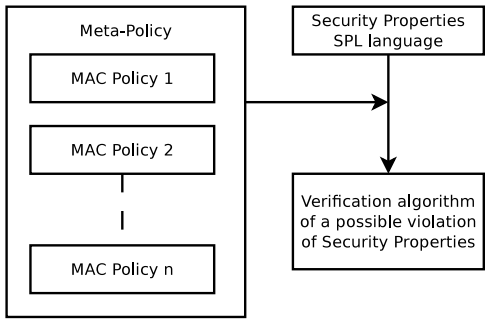


Fig. 8. Strategy of verification for dynamic policies

### D. Motivation for a verification algorithm

After presenting the meta-policy model, one can wonder if it is possible for the security properties to be enforced whereas the policies are evolving. If the policy is dynamic, that is if the administrator changes the MAC policy frequently, the generated list of activities that violate the security properties will change. In figure 7, we show that the number of MAC policies is infinite and that the strategy of enforcement of the security properties is no longer possible.

In figure 8, a solution is proposed using the meta-policy that controls all the possible policies, the computation of the infinite number of possible policies is no longer needed. We will directly use the meta-policy itself and the security properties in order to verify if a possible policy could violate one of the security properties. The goal is to get a positive or negative answer.

## VII. Verification of MAC Policies

This section presents a method to check the required security properties for dynamic MAC policies.

First, the verification of a security property can be made on a static policy (with no evolution). The algorithm is presented in section VII-A and is straightforward: it consists in a computation of paths on a graph that corresponds to the search for possible information flows, forbidden transitions, etc... The precise algorithm for each verification depends on the security property to be checked.

Second, the aim of this section is to present how to verify security properties for dynamic policies. Using the meta-policy model presented in section VI, we show how to verify a security property in order to 1) have a meta-policy that satisfies the required security properties or 2) compute all the illegal activities.

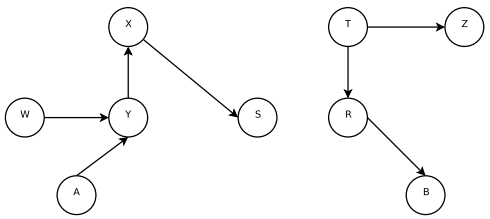


Fig. 9. Policy Graph example

For each algorithm, a concrete example for Apache is given that guarantees the integrity property of the Apache configuration files.

### A. Verification of Static Policies

As explained in section VI-C, enforcement of security properties corresponds to the computation of all the illegal paths in the flow graphs. In those graphs, our method produces all the illegal activities associated with a given security property. When deploying the MAC policy on a system, the SPLinux kernel enforces the required security properties for the local Policy [22].

### B. Verification of Dynamic Policies

Remember that the meta-policy allows the local policies to evolve. The consequences on the policy graph are detailed below, for each rule of the meta-policy:

- enable[Add|Del]SC($sc_{requester}$, $pattern_{SC}$): creates / deletes nodes in the graph labelled by $pattern_{SC}$
- enable[Add|Del]IV($sc_{requester}$, ($pattern_S$, $pattern_O$,$pattern_{oper}$)): adds / removes operations between all nodes matching $pattern_S$ and nodes matching $pattern_O$, if the operation matches $pattern_{oper}$. If no operation remains, arcs are deleted.
- enableModIV($sc_{requester}$, ($pattern_S$, $pattern_O$,$pattern_{oper}$)): modifies the operation label of arcs which joins nodes matching $pattern_S \rightarrow pattern_O$.

Consequently, enforcement is not limited to the computation of the previous static graphs because those graphs can change in the next version of the policy.

*1) Principles of the proposed method:* Let us consider the general case to check if there is an information flow between $A$ and $B$ in the graph shown in figure 9 that represents a policy. If we only consider this static graph, obviously, there is no information flow. Now let us consider the following general meta-policy rules:

Listing 23. General meta-policy rules

```
1 enableAddSC(sc_admin, expr_1,)
2 enableAddSC(sc_admin, expr_2,)
3 enableAddIV(sc_admin, expr_1, expr_2, {.*}))
```

$expr_1$ and $expr_2$ are regular expressions matching two sets of nodes. These expressions allow the administrator to

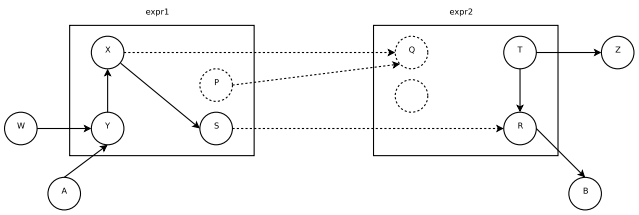


Fig. 10. Graph for listing 23

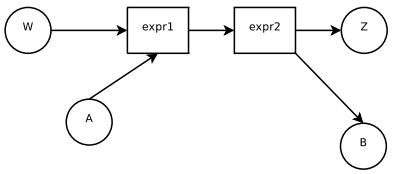


Fig. 11. Meta-nodes computation for graph of listing 23

create nodes, and modify arcs between these created nodes. Eventually, some already existing nodes can be matched by one of the regular expressions. The problem of verification of the possible existence of an information flow between $A$ and $B$ is now more complex because the administrator may add nodes and arcs and create a path between $A$ and $B$ as shown in figure 10.

We propose to solve the problem by using some extra nodes or arcs associated with the meta-policy rules. To take into account these rules, the two sets are grouped in a "meta-node" named $expr_1$ and $expr_2$. We obtain a new graph as shown in figure 11. It shows that a path can occur between $A$ and $B$: $A \rightarrow expr_1 \rightarrow expr_2 \rightarrow B$.

*2) Example:* An example for the preceding principle is given below. It is a based on the graph of the policy given in figure 5. The goal is to guarantee that there is no possible access to *apache_conf_t* for the *user_d* context when connecting via SSH. It prevents, for example, an attack that allows a user that have permission to put web pages on the web server and that exploits a vulnerability to modify or to read the Apache configuration. With the model presented in section IV, this can be written as follows:

$$\begin{cases} P_{4.1}(ssh\_d, apache\_conf\_t) & \text{(integrity)} \\ P_{4.8}(ssh\_d, apache\_conf\_t) & \text{(confidentiality)} \end{cases}$$

that is, using the SPL language:

Listing 24. Integrity and Confidentiality for the configuration of Apache
```
1  integrity( $sc1:="ssh_d", $sc2:="apache_conf_t" );
2  confidentiality( $sc1:="ssh_d", $sc2:="apache_conf_t" );
```

Obviously, with the policy of figure 5, such an attack is impossible. Nevertheless, consider the listing 25 that presents a possible meta-policy installed by the administrator. This allows the security contexts to be added for php and to add interactions between the web server and php. Special rules allow php contexts to write new information in the configuration files of Apache and in /var/www.

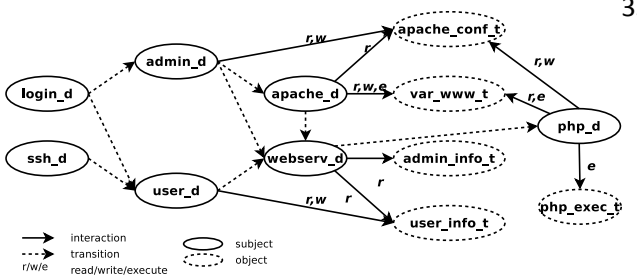


Fig. 12. Updated policy graph with rules of listing 26

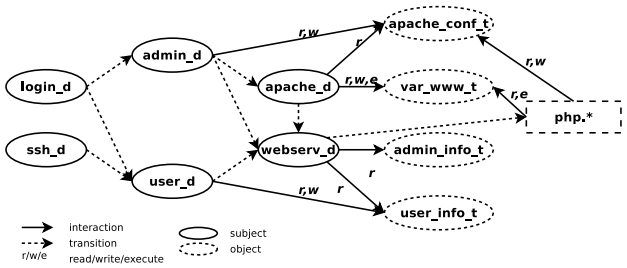


Fig. 13. Updated policy graph with rules of listing 26

Listing 25. Meta-policy for php installation
```
1  enableAddSC(admin_d, php.∗)
2  enableAddIV(admin_d, (php.∗,php.∗, {.∗}))
3  enableAddIV(admin_d, (webserv.∗,php.∗, {.∗}))
4  enableAddIV(admin_d, (php.∗,apache_conf.∗, {r,w}))
5  enableAddIV(admin_d, (php.∗,var_www.∗, {r,w,e}))
```

The listing 26 is the modification performed by the administrator that updates the policy according to the meta-policy. The figure 12 represents the new policy graph resulting from the listing 26 for the policy of figure 5. With this new policy, there is a possible attack from the *ssh_d* context against the *apache_conf_t* configuration files that was not possible before with the policy of figure 5.

Listing 26. Policy modification by admin_d user when installing php
```
1  enableSC(php_d)
2  enableSC(php_exec_t)
3  enableIV(webserv_d,php_d, {transition})
4  enableIV(php_d,php_exec_t, {e}))
5  enableIV(php_d,apache_conf_t, {r,w}))
6  enableIV(php_d,var_www_t, {r,e}))
```

The proposed methodology creates a meta-node that corresponds to the regular expression *php.**. It adds arcs 1) between this meta-node and the nodes *apache_conf_t* and *var_www_t* and 2) between *webserv_d* and the meta-node. The new graph including the meta-node is given in figure 13. In this new graph, it is easy to check if a path exists between *ssh_d* and *apache_conf_t*. The computation of this path will be reported to the administrator as a possible vector of attack.

### C. Rules with intersecting regular expressions

If we consider a meta-policy with several rules containing regular expressions, and if some of them intersect each other, then the solution presented before is not correct. This section presents a modified version of the presented methodology to take this case into account.

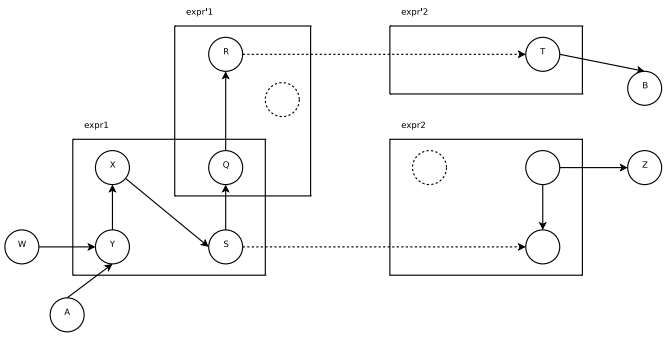


Fig. 14. Example of a meta-policy with two intersecting regular expressions

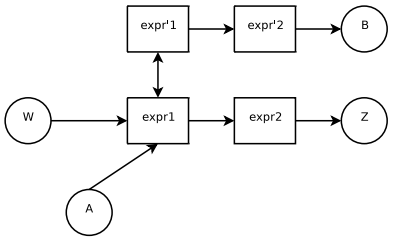


Fig. 15. Aggregated meta-graph

*1) Principles of the proposed method:* Let us suppose a general meta-policy, presented in listing 27 and represented in figure 14 where the regular expressions $expr_1$ and $expr'_1$ interleave.

Listing 27. Meta-policy rules where $expr_1$ intersects $expr_2$
```
1  enableAddSC(sc_admin, expr_1)
2  enableAddSC(sc_admin, expr_2)
3  enableAddIV(sc_admin, ( expr_1, expr_2, {.*}))
4  enableAddSC(sc_admin, expr'_1)
5  enableAddSC(sc_admin, expr'_2)
6  enableAddIV(sc_admin, ( expr'_1, expr'_2, {.*}))
```

With the preceding methodology, four meta-nodes will be built: $expr_1$, $expr'_1$, $expr_2$, $expr'_2$. Then, because of line 3 (resp. line 6) of the meta-policy of listing 27, an arc is created between $expr_1$ and $expr_2$ (resp between $expr'_1$ and $expr'_2$). But, as $expr_1$ and $expr'_1$ interleave, a possible security context $Q$ can be created inside $expr_1$ and $expr'_1$ at the same time. Thus, a new arc has to be created between $expr_1$ and $expr'_1$ as shown in figure 15.

The last remaining difficulty is to compute the possible intersections between the regular expressions. If the length of the regular expression is limited, then computing the intersection is polynomial with the number of meta-policy rules [30]. If we consider that the length of the regular expression is not limited, the algorithm cannot be computed in polynomial time [31].

*2) Example:* We now consider a more complex example of meta-policy for the installation of PHP version 5. To improve the security for php5 and to guarantee the properties $P_{4.1}(ssh\_d, apache\_conf\_t)$ and $P_{4.8}(ssh\_d, apache\_conf\_t)$ initially presented in section VII-B2, we deleted some rules, mainly to avoid the php5 process to be able to write the Apache configuration. The listing 28 shows the new meta-policy. Only three rules have been kept as we now consider that php5, in this new version, has no reason to be able to write the Apache configuration. Nevertheless, the corresponding rules have been kept in the policy file for backward compatibility for the php4 process. At this point, it is not easy to see that there is a possibility of attack from the SSH context against the Apache configuration.

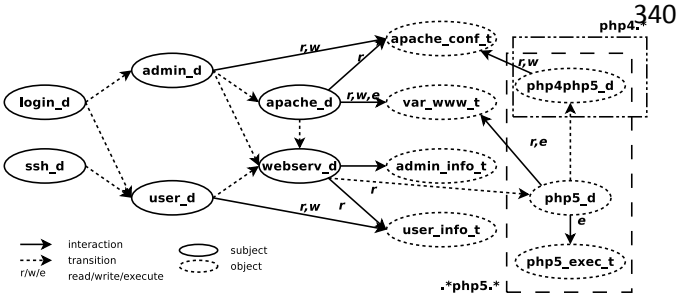


Fig. 16. Updated policy graph with rules of listing 29

Listing 28. Meta-policy for php5 installation
```
1  // Php5 meta−policy rules
2  enableAddSC(admin_d, .∗php5.∗)
3  enableAddIV(admin_d, (webserv.∗,.∗php5.∗, {.∗}))
4  enableAddIV(admin_d, (.∗php5.∗,var_www.∗, {r,w,e}))
5  // Backward compatibility rules for php4
6  enableAddSC(admin_d, php4.∗)
7  enableAddIV(admin_d, (php4.∗,apache_conf.∗, {r,w}))
```

The php5 process seems isolated from the *apache_conf_t* context. But, the administrator can create the rules of listing 29, that respect the meta-policy of listing 28. In this new policy, the *php5_d* process remains isolated from *apache_conf_t*, but a transition is possible to the context *php4php5_d*. This context matches two regular expressions: *.*php5.** and *php4.**. As the meta-policy allows *php4.** to write to the Apache configuration, the last rule of the listing allows *php4php5_d* to do it. The figure 16 shows the resulting contexts and the intersection of regular expressions appears clearly.

Listing 29. Policy modification by admin_d user when installing php
```
1  enableSC(php5_d)
2  enableSC(php5_exec_t)
3  enableIV(webserv_d,php5_d, {transition})
4  enableIV(php5_d,php5_exec_t, {e}))
5  enableIV(php5_d,var_www_t, {r,e}))
6  enableSC(php4php5_d)
7  enableIV(php5_d, php4php5_d {transition})
8  enableIV(php4php5_d,apache_conf_t, {r,w}))
```

Remember that there is a strong hypothesis in this example: the administrator creates the context *php4php5_d* for backward compatibility. It enables a malicious user to write the Apache configuration. The problem is that the administrator could not anticipate the problem when reading the listing 28, because the problem is not trivial. That is why the administrator needs a tool to analyze the meta-policy in order to check the potential violation of the security properties.

### D. Algorithm

Our algorithm computes all the possible intersections of the regular expressions. This algorithm have been implemented

in the PIGA tool and technical implementation details can be found in [22]. A set of arcs corresponding to these intersections is added in each graph (between the corresponding meta-nodes). After this first phase, each security property is processed by looking for the security pattern in the corresponding graphs. The result of this second phase is a set of illegal activities that violate the requested security property. Each illegal activity can be static, i.e. without any regular expression, or dynamic, i.e. with at least one regular expression. This second class of illegal activities is what we call a "meta-activity". The PIGA tool is able to compute any illegal activity either static or dynamic i.e a meta-activity expressed using meta-nodes that are designated with regular expressions.

**Require:** G: the interaction graph i.e. the initial policy
**Require:** $MPR_{sc}$: meta-policy rules of type EnableAddSC
**Require:** $MPR_{iv}$: meta-policy rules of type EnableAddIV
**Ensure:** G': the computed meta-graph

1: G'=G.clone()
2: **for** $sc \in MPR_{sc}$ **do**
3: G'.addNode(new Node(sc))
4: **end for**
5: **for** each $rule \in MPR_{iv}, rule = enableAddIV(s,(expr_1, expr_2, perms))$ **do**
6: G'.addNode($v_{expr_1}$ = new Node($expr_1$))
7: G'.addNode($v_{expr_2}$ = new Node($expr_2$))
8: G'.addArc($v_{expr_1}$, $v_{expr_2}$)
9: G'.addArc($v_{expr_2}$, $v_{expr_1}$)
10: **for** each $v \in G'$ **do**
11: **if** $v$ matched by $expr_1$ **then**
12: **for** each $v' \in G'.neighbors(v)$ **do**
13: G'.addArc(v', $v_{expr_1}$, G'.dir(v',v))
14: **end for**
15: **end if**
16: **if** $v$ matched by $expr_2$ **then**
17: **for** each $v' \in G'.neighbors(v)$ **do**
18: G'.addArc(v', $v_{expr_2}$, G'.dir(v',v))
19: **end for**
20: **end if**
21: **end for**
22: **end for**
23: **for** each $rule \in MPR_{iv}, rule = enableAddIV(s,(expr_1, expr_2, perms))$ **do**
24: **for** each $rule' \in MPR_{iv}, rule = enableAddIV(s,(expr_3, expr_4, perms))$ **do**
25: **if** $expr_1 \cap expr_3 \neq \emptyset$ **then**
26: **if** $\exists v \in G'$ / v matches $expr_1$ and $expr_3$ **then**
27: G'.addArc($v_{expr_1}$, $v_{expr_3}$)
28: G'.addArc($v_{expr_3}$, $v_{expr_1}$)
29: **end if**
30: **end if**
31: // Same treatment for ($expr_1$, $expr_4$), ($expr_2$, $expr_3$), ($expr_3$, $expr_4$)
32: **end for**
33: **end for**

This algorithm computes the new meta-graph G' that is used to search paths between two security contexts. The algorithm first creates the meta-nodes associated to the regular expressions into G' (lines 1-9). Then, if the nodes of G' can be included in one of the regular expressions, the arcs are updated to link the meta-nodes (i.e. the regular expressions) to the corresponding neighbours (lines 10-22). In other words, all the matching nodes are grouped within the corresponding meta-nodes. The second part of the algorithms checks if two meta-nodes can intersect each other in which case the two meta-nodes are linked together.

## VIII. Experimentation on a Honeypot

In order to secure a high-interaction Honeypot, we defined the security properties of listing 30 in order to give high protection to the system resources and prevent the system from being corrupted. The precise experiment, the technical details, and the discussion about the results are given in [4]. Some of the results are recalled in this section to give to the reader an illustration of the use of the security properties modelled using the SPL language.

Several standard Linux hosts without special protection have been setup to see if attackers can compromise them. Those kind of hosts are compromised in less than a week. Windows hosts are surviving only one day. The compromised hosts have been removed from the experiment after some weeks because it is too much work to maintain and monitor them. In contrast with these standard Linux and Windows hosts, the SPLinux protected honeypot systems have never been compromised during two years of experiment.

The protected hosts that are considered in the presented results are the one that have been analyzed by the PIGA tool [22] which formal algorithm has been described in section VII. The protection is then enforced using our SPLinux kernel which monitors in real time the activities of the system processes. If a process violates a security property, the SPLinux kernel prevents the execution of an interaction of the incriminated activity.

The first property, *integrity*, expresses the fact that no subject context can modify the binary files. The second rule, *confidentiality*, prevents the user from reading any file of the system. The third property, *int_domain*, expresses the fact that no chrooted context is able to interact with any context outside the chrooted domain. The fourth property, *no_transition*, expresses the fact that a system process cannot transit to a bad domain (a blacklist of dangerous contexts). At last, the third property, *duties_separation*, expresses the fact that a context is not able to be executed by another one if it has only just been modified by this same another one.

Listing 30. Security properties for the honeypot

```
1 integrity($sc1:=".*", $sc2=".*:.*:.*_exec_t");
2 confidentiality($sc1:=user_u:user_r:user_t, $sc2:="system_u:object_r
      :.*" );
3 int_domain( $CHR:=".*:.*:.*user.*");
4 no_transition($sc:=user_u:user_r:user_t);
5 duties_separation($sc1:=".*");
```

TABLE III
ILLEGAL META-ACTIVITIES.

| | | Gateway | User | Initial |
|---|---|---|---|---|
| Graph | $SC$ | 577 | 3 017 | 595 |
| | $IV$ | 17 684 | 314 582 | 18 215 |
| Security Property Rules | integrity | 137 | 9 461 | 140 |
| | int_domain | 16 283 | 510 215 | 16 546 |
| | confidentiality | 29 510 | 726 842 | 29 510 |
| | duties_separation | 243 | 16 405 | 270 |
| | no_transition | 3 555 | 126 228 | 3 941 |
| Results | Size of the database | 1,1MB | 3,6MB | 1,1MB |
| | Number of audit | 13 664 | 44 503 | 14051 |
| | Computation time | 47s | 10min31s | 52s |

Let us consider those properties using several complete meta-policies for two kinds of host. The first kind of host is a gateway and the second kind is a user host.

Each host contains the same initial policy as a part of the meta-policy. That initial policy is processed as a static policy. The results are given in column $Initial$. The PIGA tool finds 140 activities that can violate the $integrity$ property, 16,546 activities violating the $int\_domain$ property, 29,510 activities violating the $confidentiality$ property, 270 activities violating the $duties\_separation$ property and 3.941 activities violating the $no\_transition$ property.

The meta-policy of the gateway host includes few modification rules. The PIGA tool processes the corresponding meta-activities, including 137 activities that can violate the $integrity$ property, 16,283 activities violating the $int\_domain$ property, 29,510 activities violating the $confidentiality$ property, 243 activities violating the $duties\_separation$ property and 3,555 activities violating the $no\_transition$ property. The resulting meta-graphs are smaller than the initial graph since the update rules of the gateway permit some security contexts and interactions to be removed. Thus, it is consistent to have fewer meta-activities than initial activities.

The meta-policy of the user host includes much larger modification rules. PIGA computes 9,461 activities that can violate the $integrity$ property, 510,215 activities violating the $int\_domain$ property, 726,842 activities violating the $confidentiality$ property, 16,283 activities violating the $duties\_separation$ property and 126,228 activities violating the $no\_transition$ property. The user host authorizes many updates of the initial policy in order to support all the classical applications such as the Gnome desktop, Firefox, OpenOffice, etc. The majority of the updates enable new security contexts and interactions to be added for those applications. Thus, PIGA finds many meta-nodes authorizing the user contexts to interact with the system. Since all the properties aim at limiting the flows between the user and the system, it is consistent to have many more illegal meta-activities between the system and the user.

In addition, Table III shows the size of the database of the corresponding illegal activities plus the time needed for computation. At present, computation is not optimal since the solution computes each meta-policy as an independent set of rules. Optimizations could be provided by reusing the results already obtained for previous meta-policies. Finally, the resulting illegal activities associated with the permitted updates of the initial policy can be extracted from the database. Thus, the administrator can isolate the illegal activities that are permitted by the modification constraints. He can either modify the modification constraints or decide that it is consistent. In the later case, each time a new local policy is set up, the corresponding illegal activities will be prevented by our SPLinux kernel. Thus, a satisfactory solution is provided for dynamic MAC policies.

## IX. CONCLUSION

This paper introduces a precise formalization of a wide range of integrity and confidentiality properties. A dedicated language has been developed to describe these properties. The abstract Security Property Language (SPL) deals with the activities of the operating system and models complex correlations between these activities that can lead to illegal activities. Even if it easy to define abstract properties, the paper gives 13 pre-defined security templates. Some of them are properties well known in the literature, but others are new security properties.

Moreover, a concrete SPL language is proposed that gives a concrete syntax for enforcing the required properties. Thus, concrete security properties are supported for the predefined template. New templates can be easily defined using the concrete SPL language. A compiler is available for processing mandatory access control policies such as SELinux policies. That compiler computes all the illegal activities that could violate the requested properties. The paper demonstrates how to write concrete properties for protecting systems.

This paper also shows how the security properties are linked to mandatory access control mechanisms. The security properties and the MAC policy of the target host allow the compiler to generate the list of illegal activities. This list is the input of our SPLinux kernel that will enforce the security properties. Thus, a system call fails if illegal activities are recognized.

In the case of dynamic policies, the list of illegal activities is not computable. The paper shows that, with the use of a meta-policy, a policy that constraints the MAC policies, it becomes possible to verify if the meta-policy violates one of the security properties. The verification algorithm is based on a special graph where meta-nodes and meta-arcs are added in accordance with the meta-policy. The result of the algorithm is that the administrator knows if the meta-policy authorizes a policy that violates one of the defined security properties. It is a strong result as it is difficult to be able to give such a guarantee for dynamic policies. Moreover, with a simple meta-policy, all the possible illegal activities can be computed in order to improve the meta-policy.

Finally, the paper gives results about the computation of the illegal activities that violate security properties for two kind of hosts of our honeypot. A large variety of combinations of interactions allow the security properties to be violated. Nevertheless, these properties are enforced by our SPLinux kernel. SPLinux prevents the honeypot hosts from being compromised. During two years of experiments, our honeypot systems have never been compromised whereas a standard linux host have been always compromised in less than one week. It is this experiment that shows the efficiency of our approach. Thus, a global solution is provided. Verification is not mandatory since the administrator can trust the SPLinux kernel to enforce all the required security properties. However, verification is a powerful tool to adjust both the meta-policy or the required security properties.

Future works deal with optimization of our compiler to reuse the verification methods. Indeed, the illegal activities can be pre-computed for all the permitted meta-nodes. Thus, a complete database can be provided for enforcing the host. That database will include both static and dynamic activities to prevent the compilation phase when a local policy is updated. Moreover, ongoing works address DAC systems. Despite the fact that DAC systems are more complicated to protect, we are developing a new enforcement mechanism for those kinds of host. Thus, the security properties will be enforced for both DAC and MAC systems. Finally, extensions will be proposed to enable the SPL language to deal with availability and distributed properties.

## REFERENCES

[1] J. Briffaut, J.-F. Lalande, C. Toinard, and M. Blanc, "Enforcement of security properties for dynamic mac policies," in *Third International Conference on Emerging Security Information, Systems and Technologies*, IARIA, Ed. Athens/Glyfada, Greece: IEEE Computer Society Press, June 2009, pp. 114–120.

[2] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman, "Protection in operating systems," *Communications of the ACM*, vol. 19, no. 8, pp. 461–471, 1976.

[3] M. Blanc, J. Briffaut, J.-F. Lalande, and C. Toinard, "Distributed control enabling consistent MAC policies and IDS based on a meta-policy approach," in *Proceedings of the Seventh IEEE International Workshop on Policies for Distributed Systems and Networks*. London, Canada: IEEE Computer Society, Jun. 2006, pp. 153–156.

[4] J. Briffaut, J.-F. Lalande, and C. Toinard, "Security and results of a large-scale high-interaction honeypot," *Journal of Computers, Special Issue on Security and High Performance Computer Systems*, vol. 4, no. 5, pp. 395–404, may 2009.

[5] F. Schneider, "Enforceable security policies," *Information and System Security*, vol. 3, no. 1, pp. 30–50, 2000.

[6] R. Focardi and S. Rossi, "Information flow security in dynamic contexts," in *Proceedings of the IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 2002, pp. 307–319.

[7] M. Giunti, "Preventing intrusions through non-interference," in *Proceeding of the IEEE Mexican Conference on Informatics Security*. IEEE Computer Society Press, 2006.

[8] C. Ko and T. Redmond, "Noninterference and intrusion detection," in *IEEE Symposium on Security and Privacy*, 2002, pp. 177–187.

[9] G. Hiet, V. Viet Triem Tong, B. Morin, and L. Me, "Monitoring both os and program level information flows to detect intrusions against network servers," in *Proceedings of the 2nd workshop on MONitoring, Attack detection and Mitigation*, Nov. 2007.

[10] G. Hiet, V. V. T. Tong, and L. Mé, "Policy-based intrusion detection in web applications by monitoring java information flows," in *CRISIS '08: 3nd International Conference on Risks and Security of Internet and Systems*, Oct. 2008.

[11] S. Zdancewic, "Challenges for information-flow security," in *Proceedings of the 1st International Workshop on the Programming Language Interference and Dependence*, 2004.

[12] V. C. Hu, E. Martin, J. Hwang, and T. Xie, "Conformance checking of access control policies specified in xacml," in *Proceedings of the 31st Annual International Computer Software and Applications Conference*, vol. 2. Washington, DC, USA: IEEE Computer Society, 2007, pp. 275–280.

[13] G.-J. Ahn, W. Xu, and X. Zhang, "Systematic policy analysis for high-assurance services in selinux," in *Proceedings of the 2008 IEEE Workshop on Policies for Distributed Systems and Networks*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 3–10.

[14] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières, "Making information flow explicit in histar," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2006, pp. 19–19.

[15] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris, "Information flow control for standard os abstractions," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 321–334, 2007.

[16] P. Efstathopoulos and E. Kohler, "Manageable fine-grained information flow," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 4, pp. 301–313, 2008.

[17] M. L. Damiani, C. Silvestri, and E. Bertino, "Hierarchical domains for decentralized administration of spatially-aware rbac systems," in *ARES '08: Proceedings of the 2008 Third International Conference on Availability, Reliability and Security*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 153–160.

[18] L. Seitz, E. Rissanen, T. Sandholm, B. S. Firozabadi, and O. Mulmo, "Policy administration control and delegation using xacml and delegent," in *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 49–54.

[19] N. Li, Z. Mao, and H. Chen, "Usable mandatory integrity protection for operating systems," in *IEEE Symposium on Security and Privacy*, Berkeley, California, May 2007, pp. 164–178.

[20] Z. Mao, N. Li, H. Chen, and X. Jiang, "Trojan horse resistant discretionary access control," in *Proceedings of the 14th ACM symposium on Access control models and technologies*. New York, NY, USA: ACM, 2009, pp. 237–246.

[21] X. Cai, Y. Gui, and R. Johnson, "Exploiting unix file-system races via algorithmic complexity attacks," in *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 27–41.

[22] J. Briffaut, J. Rouzaud-Cornabas, C. Toinard, and Y. Zemali, "A new approach to enforce the security properties of a clustered high-interaction honeypot," in *Workshop on Security and High Performance Computing Systems*, R. K. Guha and L. Spalazzi, Eds. Leipzig, Germany: IEEE Computer Society, June 2009, pp. 184–192.

[23] ITSEC, "Information Technology Security Evaluation Criteria (ITSEC) v1.2," Technical Report, Jun. 1991.

[24] K. J. Biba, "Integrity considerations for secure computer systems," The MITRE Corporation, Technical Report MTR-3153, Jun. 1975.

[25] R. Focardi and R. Gorrieri, "Classification of security properties (part I: Information flow)," in *Foundations of Security Analysis and Design*. Springer Berlin / Heidelberg, 2001, pp. 331–396.

[26] C. Ko and T. Redmond, "Noninterference and intrusion detection," in *IEEE Symposium on Security and Privacy*. Berkeley CA, United-States: IEEE Computer Society, May 2002, pp. 177–187.

[27] D. E. Bell and L. J. La Padula, "Secure computer systems: Mathematical foundations and model," The MITRE Corporation, Bedford, MA, Technical Report M74-244, May 1973.

[28] D. D. Clark and D. R. Wilson, "A comparison of commercial and military computer security policies," in *The Symposium on Security and Privacy*. IEEE Press, 1987, pp. 184–193.

[29] R. Sandhu, "Separation of duties in computerized information systems," in *Database Security, IV: Status and Prospects*, Halifax, U.K., September 1990, pp. 179–190.

[30] K. Dexter, "Lower bounds for natural proof systems," in *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*. Washington, DC, USA: IEEE Computer Society, 1977, pp. 254–266.

[31] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*. Addison Wesley, November 2000.