An Information Flow Approach for Preventing Race Conditions: Dynamic Protection of the Linux OS

Jonathan Rouzaud-Cornabas, Patrice Clemente, Christian Toinard
ENSI de Bourges – Laboratoire d'Informatique Fondamentale d'Orléans
88 bd Lahitolle, 18020 Bourges cedex, France
{jonathan.rouzaud-cornabas,patrice.clemente,christian.toinard}
@ensi-bourges.fr

***Abstract*—In the literature, the notion of Race Condition deals with the interference between two processes A and B carrying out three interactions involving a shared object. The second interaction of the concurrent process B interleaves with the first and the third interactions of process A. Preventing Race Conditions attacks between concurrent processes is still an open problem. Many limitations remain such as preventing only Race Conditions on a file system or controlling only direct interactions with the shared context. That paper covers those various problems. First, it gives a formal definition of direct and indirect information flows at the scale of a complete operating system. Second, it proposes a general formalization of Race Conditions using those information flows. In contrast with existing formalizations, our definition is very effective and can be implemented on any operating system. Third, it provides a Mandatory Access Control that enables to prevent general Race Conditions at the scale of a whole Linux operating system. The Race Conditions can be easily expressed as a Security Properties policy. A honeypot experimentation provides a large scale evaluation of our dynamic MAC enforcement. It shows the efficiency to prevent both direct and indirect Race Conditions. Performances are encouraging us to follow our approach of a dynamic MAC for enforcing a larger range of security properties.**

***Keywords*-Computer security; data security; access control; operating systems.**

I. Introduction

A Race Condition (*RC*) happens when there is an unpredictable schedule between accesses of two users or processes to a conflicting resource, having at least one of the two users/processes modifying the shared resource. Depending on the scheduling of the accesses, the content of the resource may be unexpected. Historically, attacks based on *RC* used some unpredictable behaviors of OS (e.g., with signals) in order to modify the behavior of legitimate processes for example. The time-of-check-to-time-of-use (TOCTTOU) flaw happens when a process checks the attributes of a file, and performs operations on it assuming that the attributes have not changed when actually they have. In the literature, some authors have proposed formal description of *RC*s but provide only partial or no implementation at all. Other work deals with *RC*s within parallel programs. Remaining work focuses on protecting against *RC*s attacks but only specific ones, e.g., TOCTTOU. Many limitations remain, and in practice only a partial cover of this problem is proposed.

This paper is an extended version of [1]. In this paper we propose a global approach to deal with *RC*s at the scale of a complete computer system, at the operating system level. We provide a formal modeling of OS, and a definition of information flows related to any system call available on OS. That definition enables to formalize general *RC*s at the scale of a whole operating system. It considers a general information flow including both direct information flows and indirect information flows. General information flows are more difficult to prevent since they can involve several processes and resources using covert channels. A definition of *RC*s is given, based on three general information flows between concurrent processes. This definition is provided using a general framework for the definition of any security property related to information flows. A dynamic Mandatory Access Control is proposed to enforce the required properties of *RC*s within the Linux kernel. An experimentation is presented with several honeypot hosts exposed to the Internet including well known vulnerabilities. It shows that our MAC approach correctly prevented the *RC*s attacks. Finally, a performance study shows the real efficiency of our implementation.

The paper is organized as follows. Next section details existing work and motivates the paper. Section III gives a formal definition of the operating system and information flows. In Section Section III-D is presented our general definition of *RC*s, followed in Section III-E with the description of our linux kernel module PIGA-DYNAMIC-PROTECT. Section IV presents experiment results. Lastly, Section V gives performance evaluations, before concluding the paper, in Section V-B.

II. Related Work

In this section, we first present the state of the art of race condition. Then, we do a scope of the different protection mechanisms that have been proposed for operating system. Finally, we explain our motivation.

A. Race Condition

The authors of [2] have proposed an informal definition of a race condition:

Definition 2.1 (Informal definition of a race condition): A race condition happens when there is an unpredictable

schedule between accesses of two subjects X and Y to a shared resources I, having at least one of the two former subjects performing a write operation to the shared object between two accesses of the other subject to the shared object.
Thus, the state of the system depends on the execution order and thus is unpredictable.

It exists different types of race condition: on data [3], on signals [4] and on any shared resources. The more common case of race condition is the one associated with preliminary checks done by applications on resources. This race condition is known as time-of-check-to-time-of-use (TOCTTOU) and has been defined in [5]. Most of the filesystems are vulnerable to such attack [6], [7]. Moreover, with the increase of processors' cores within a single operating system, the detection of (and protection against) race condition becomes more and more complex. Indeed, within this scope, the detection becomes a NP-Complete problem [8].

Four main approaches are used to protect against race conditions:

- Code analysis: By analyzing the code before compiling it, it is possible to detect and find parts of code that could lead to race condition flaws [7], [9], [10], [11];
- Dynamic detection: By auditing system calls that could be used to build a race condition attack, it is possible to detect them. The most known approach is the one by Ko and Redmond [12] that detects race conditions but are not able to protect against them. Other approaches [13], [14] have used the same idea;
- Dynamic protection: When the concurrency is detected, it is possible to kill or suspend the corresponding process or system call. The authors of [15], [16], [17] have proposed this kind of approach to protect a system.
- Filesystems: It is possible to build filesystems that take into account race conditions and concurrency. Two kinds of such filesystems have been proposed: one using transactional filesystems [18], [19] and one with system calls designed to deal with concurrency [20], [21].

But, except fully transactional systems, no approach proposes a complete protection against race conditions. Transactional approaches misfit for operating systems. Moreover, the previous approaches can not express explicitly which race condition to control and do not deal with all the kinds of race condition.

B. Operating System Protection

Two main models of protection are currently used in operating systems. They are used to control the access of the users on files and others objects based on privileges. The Role Based Access Control [22] (RBAC) does not change anything in term of security, it eases the writing of security policies.

The Discretionary Access Control (DAC) is the oldest protection model. But it is always the main access control model used in modern operating system (Unix, MS Windows, Mac OSX, ...). With DAC, the privileges are set by the user who owns the object. For example, the owner of a file defines the read, write, execute privileges on its files for the other users on the system (himself, the ones within his group and the others). Multiple studies [23], [22], [24] have shown the weakness of DAC models. Indeed, it is based on the fact that users can set efficiently the permissions on their files. But any errors can lead to a security flaw. For example, if the users password file can be written by any users, anyone with an account on the operating system can change super-user password and thus obtain its privileges.

The Mandatory Access Control (MAC) allows to setup a policy that can not be change by end-users. To monitor the access between subjects and objects, Anderson [25] has proposed to use a Reference Monitor. This concept is the base of Mandatory Access Control. It was defined for Bell&LaPadula approach [26] but is now used to describe any mechanism that put the management of the security policy outside the scope of end-users. To ease the writing of security policies for operating system, it is needed to associate each entity (subject and object) with a type. This approach of Type Enforcement has been introduced in [27]. It facilitates the definition and aggregation of security model. But the drawback of the MAC approach is the difficulty to define efficient security policies. Indeed, you need to have an in-depth knowledge of how work the operating system (and its applications) and the security objectives that you want to reach. It is the cause of the low usage of the MAC approach (GRSecurity, SELinux) in modern operating systems.

Approaches like [28], [29], [30] try to bring the ease of DAC with the protection of MAC. They state that: "a good enough and usable policy is better than a very good security but hard to manage". But it is dedicated to the protection of a desktop operating system from network attacks. Moreover, they can not express security models and the quality of their protection is questionable as they does not take into account indirect flows and based their models on DAC permissions.

Others approaches [31], [32] are oriented toward the control of information flow. The purpose is to isolate users from each others. In this kind of approach, the security is enforced by a reference monitor within the system but the policy are written by the application developers. Indeed, [31] states that developers are best to know which security is needed by their applications. Moreover, the protection is done by the operating system with a reference monitor in the kernel. Thus, even if the application contains flaws, it respects the security policy. This model has been extended to GNU/Linux with the Flume [33] framework. They also proposes a limited language to describe the information flow control policy. But all those approaches can not combine flows. Thus, they are limited to describe simple security models. Moreover, they request to rewrite part of the applications' code. The authors of [34] have modelized Flume to prove that it does not contain covert channels. It does not prove the expressiveness of their language but the dependability of their system.

C. Motivation

First, we have explain the need of a mandatory approach to avoid the weakness of a discretionary control. But, current mandatory models do not explicitly take into account transitive flows and are hard to implement on a real operating system. Second, the existing protections of operating systems do not propose a language to express security objectives. Thus, they can not be used to express a complete formal security policy. The approaches based on the control of the information flows are difficult to use and do not propose a language to express security properties.

Our goal is to ease the definition of any security properties in order for the operating system to guarantee the requested properties. In this paper, we focus on race conditions but others properties have been already defined [35]. The proposed security property for protecting against race conditions needs to take into account all kinds of race conditions (filesystems, signals, any kind of data, etc). This property is formally expressed. It explicitly defines the subjects that are protected. Moreover, our language takes into account the dynamicity of a real operating system i.e. its state can change at anytime. The dynamicity is very important as we want to be able to implement our language and our protection on existing operating systems. The ability to implement our proposal is important as it is a mean to provide a large scale of real world experimentations.

III. System and security properties modeling

In order to formalize the security property that protects against *RC* attacks, in terms of activities on the operating system, let us first define the model of the target system. The first requirement is to be able to associate a unique security label (also called security context) to each system resource. A security context can be a file name or the binary name executed by a process. Our system fits well for DAC OS (GNU Linux, Microsoft Windows) or MAC ones such as SELinux whereas security contexts are special entities controlled by the kernel.

A. System dates, entities and operations

In essence, an operating system is defined by a set of entities performing operations on other entities. Those entities are referred here as 'security contexts'. Acting contexts are called subject contexts while passive ones are called object contexts.

Formally, an operating system consists of the following elements:

- A set of system dates $\mathcal{D}$.
  Any $d \in \mathcal{D}$ is a number representing a system date.
- A set of subject security contexts $\mathcal{SSC}$.
  Each $ssc \in \mathcal{SSC}$ characterizes an active entity, i.e. processes, that can perform actions, i.e. system calls.
- A set of object security contexts $\mathcal{OSC}$.
  Each $osc \in \mathcal{OSC}$ characterizes a passive entity (file, socket, ...) on which system calls can be performed.
- A set of all security contexts $\mathcal{SC} = \mathcal{SSC} \cup \mathcal{OSC}$, with $\mathcal{SSC} \cap \mathcal{OSC} = \emptyset$.
  For example, let us consider the apache webserver reading an HTML file. The apache process is identified as a subject ($/usr/bin/apache \in \mathcal{SSC}$ in a classical Linux system or $apache\_t \in \mathcal{SSC}$ in a SELinux environment) and the file is considered as an object ($/var/www \in \mathcal{OSC}$ in a classical Linux system or $var\_www\_t \in \mathcal{OSC}$ in a SELinux environment).
- A set of elementary operations $\mathcal{EO}$.
  $\mathcal{EO}$ denotes all the elementary operations, i.e. system calls, that can occur on the system (i.e. $read\_like$ and $write\_like$ operations).
- A set of interactions : $\mathcal{IT} : \mathcal{SSC} \times \mathcal{EO} \times \mathcal{SC} \times \mathcal{D} \times \mathcal{D}$.
  Each element of $\mathcal{IT}$ is thus a 5-uple that formally represents an interaction on the system. In essence, an interaction $it \in \mathcal{IT}$ represents a subject $ssc \in \mathcal{SSC}$ invoking an operation $eo \in \mathcal{EO}$ on a given context $tsc \in \mathcal{SC}$, starting at a system date s and ending at a system date e.
- A system trace T.
  The execution of an operating system can be seen as a set of invoked interactions. The executed interactions modify the OS state [17]. When we consider prevention, we work with an invocation trace. The invocation trace contains thus all tried interactions, even those which are finally not allowed to be executed. Thus, each time an interaction it_i occurs on a given system (before being allowed, in case of prevention system), the corresponding system trace becomes $T_i \leftarrow T_{i-1} \cup it_i$.

B. Information Flows

1) Direct Information Flows: In terms of information flows, when an interaction occurs (i.e. an elementary operation is performed), there is one potential consequence: that interaction can produce an information flow from one security context to another.

An *information flow* transfers some information from a security context sc_1 to a security context sc_2 using a $write\_like$ operation or to sc_1 from sc_2 using a $read\_like$ operation[1].

The formal modeling of the system is then extended with the following sets:

- A subset of $\mathcal{EO}$ of $read\_like$ operations $\mathcal{REO}$.
- A subset of $\mathcal{EO}$ of $write\_like$ operations $\mathcal{WEO}$.

Definition 3.1 (Direct Information Flow): Given a system trace T, a direct information flow from a subject context ssc performing a $write\_like$ operation to a target context tsc, starting at a system date s and ending at a system date e (formally an interaction: (ssc, weo, tsc, s, e), where $ssc \in \mathcal{SSC}, tsc \in \mathcal{SC}, weo \in \mathcal{WEO}, s \in \mathcal{D}, e \in \mathcal{WEO}$, and $s \leq e$), is denoted by: $ssc \overset{T}{\triangleright}_{[s,e]} tsc$.

Symmetrically, a direct information flow from a subject context ssc performing a $read\_like$ operation to a target

[1]To be able to decide if an interaction produces an information flow between two security contexts, we use a mapping table (not detailed here) that says for each $eo \in \mathcal{EO}$ if it can flow information – and in what direction between the two security contexts – or not.

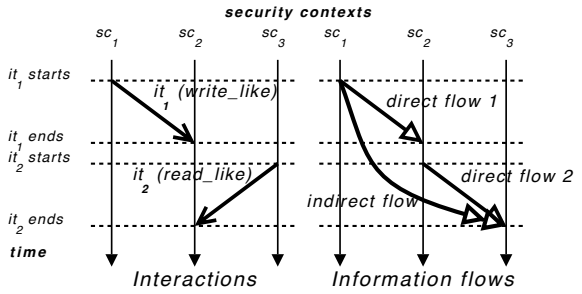

Fig. 1. r/w_like interactions and corresponding information flows.

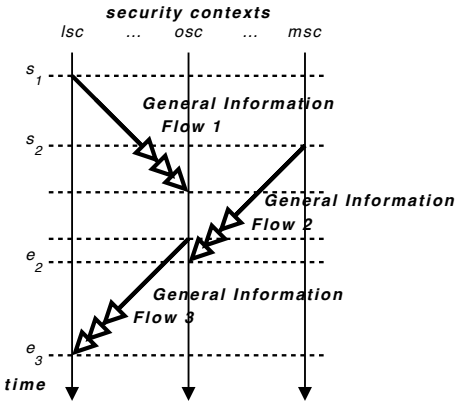

Fig. 2. Example of *RC* between lsc and msc.

context tsc starting at s and ending at e is denoted by: $tsc \overset{T}{\triangleright}_{[s,e]} ssc$.

Notice that we use the following abuses of notation when respectively s, e or both are not explicitly required: $\overset{T}{\triangleright}_{[\_,e]}$, $\overset{T}{\triangleright}_{[s,\_]}$, $\overset{T}{\triangleright}$.

2) Indirect Information Flows: As said previously, an information flow can occur directly between two security contexts. But it can also happen in many indirect ways. For example, there can exist a first flow $ssc \overset{T}{\triangleright} osc$ and then a second flow $osc \overset{T}{\triangleright} tsc$. We consider this as an *indirect information flow* from ssc to tsc. Transitively, there can theoretically be an infinite number of intermediary contexts between ssc and tsc.

We propose the following definition of indirect information flows:

Definition 3.2 (Indirect Information Flow): Given a system trace T, an indirect information flow from one context sc_1 to another context sc_k, starting at a system date s_1 and ending at a system date e_k, denoted by $sc_1 \overset{T}{\blacktriangleright\!\!\blacktriangleright}_{[s_1,e_k]} sc_k$ (i.e., k is the number of contexts involved in the indirect flow), is formally defined by:

$$\left(\begin{array}{c} \exists k \in [3..+\infty], \forall i \in [1..k-2], sc_i \in \mathcal{SC}, \{s_i, e_i\} \in \mathcal{D}, \\ (sc_i \overset{T}{\triangleright}_{[s_i,\_]} sc_{i+1}) \wedge (sc_{i+1} \overset{T}{\triangleright}_{[\_,e_{i+1}]} sc_{i+2}) \\ \wedge (s_i \leq e_{i+1}) \end{array} \right)$$

The Figure 1 shows an example of such an indirect information flow where $k = 3$. The first interaction it_1 is a $write\_like$ operation from sc_1 to sc_2. The second interaction it_2, is a $read\_like$ operation of sc_3 to sc_2. Thus, sc_3 gets information from sc_2. So, there is an indirect information flow between sc_1 to sc_3 via the shared context sc_2.

3) General Information Flows: The two previous definitions lead to a more general definition of information flows. In essence, there exists an information flow between two security contexts *iff* there exists a direct flow or an indirect flow between those contexts.

Definition 3.3 (General Information Flow): Given a system trace T, an information flow from one context sc_1 to another context sc_k, starting at the system date s_1 and ending at the system date e_k, denoted by $sc_1 \overset{T}{\blacktriangleright\!\!\blacktriangleright}_{[s_1,e_k]} sc_k$, is formally defined by:

$$(sc_1 \overset{T}{\triangleright}_{[s_1,e_k]} sc_k) \vee (sc_1 \overset{T}{\blacktriangleright\!\!\blacktriangleright}_{[s_1,e_k]} sc_k)$$

Again, by abuse of notation, $\overset{T}{\blacktriangleright\!\!\blacktriangleright}_{[\_,e]}$, $\overset{T}{\blacktriangleright\!\!\blacktriangleright}_{[s,\_]}$, and $\overset{T}{\blacktriangleright\!\!\blacktriangleright}$ is used when s, e or both are not explicitly required.

C. Race Condition

As presented earlier, [2] gave a general definition of *RC* that we express here under our formalism in order to be able to define an enforceable security property to prevent *RC* attacks.

Definition 3.4 (General Race Condition): A *RC* happens when there is an unpredictable schedule between accesses of two security contexts sc_1 and sc_2 to a conflicting data context osc (i.e. a shared security context), having at least one of the two former contexts (e.g., sc_1) performing a $write\_like$ operation to the shared context osc between two accesses of the other context (e.g., sc_2) to the conflicting data context osc.

D. Race Condition Security property

Using the General *RC* definition above, and in order to detect or prevent *RC* based attacks, we propose to define a general Security Property for the prevention of *RC* attacks.

Security Property 3.1: A security context lsc is protected against a *RC* from another security context msc *iff* msc can not transfer information to a shared context osc between two accesses of lsc to this shared context osc.

Formally: $\text{No\_Race\_Condition}(lsc, msc, T) \Leftrightarrow$

$$\neg \left(\begin{array}{cc} \exists osc \in \mathcal{SC} \wedge & \\ \big((lsc \overset{T}{\blacktriangleright\!\!\blacktriangleright}_{[s_1,\_]} osc) \vee (osc \overset{T}{\blacktriangleright\!\!\blacktriangleright}_{[s_1,\_]} lsc)\big) \wedge & (1) \\ (msc \overset{T}{\blacktriangleright\!\!\blacktriangleright}_{[s_2,e_2]} osc) \wedge & (2) \\ \big((lsc \overset{T}{\blacktriangleright\!\!\blacktriangleright}_{[\_,e_3]} osc) \vee (osc \overset{T}{\blacktriangleright\!\!\blacktriangleright}_{[\_,e_3]} lsc)\big) \wedge & (3) \\ \big((s_1 \leq e_2) \wedge (s_2 \leq e_3)\big) & (4) \end{array} \right)$$

In the security property definition above, lsc typically represents a legitimate security context while msc represents a potentially malicious (attacker's) context.

There are many temporal situations covered by this definition, including partially or totally concurrent situations.

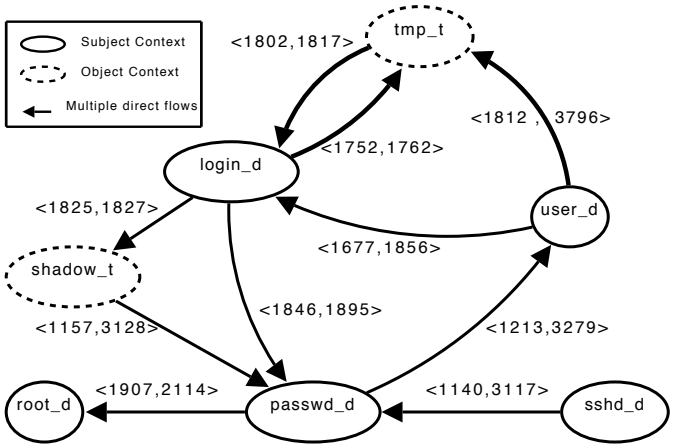

Fig. 3. IFG for the login *RC* scenario.

The Figure 2 shows one of those possible schedules. The first information flow represents an access from a legitimate context lsc to a shared context osc. It corresponds to the line #1 of the $\mathrm{No\_Race\_Condition}$ security property. The second information flow corresponds to line #2 of the property. It represents a flow from a malicious context msc to osc after or concurrently to the first flow. The third flow on the Figure corresponds to the line #3 of the property. It represents the second access from lsc to osc (partially) after or during the modification of osc by msc. The line #4 of the property expresses temporal constraints between the flow, for sequential or concurrent situations.

E. Enforcing the RC security property

In contrast to [2], this definition, based on information flows, allows us to provide an effective and efficient algorithm and related implementation for the protection against *RC* attacks.

Our solution computes an Information Flow Graph (IFG). An example of an IFG is given in Figure 3. The IFG manages the temporal relationships between the security contexts using one parameter on each edge connecting two security contexts. This parameter is a couple of system dates of the first and last occurrences of the represented information flow. Actually, those are the dates of $read\_like$ or $write\_like$ interactions.

Thus, an edge on the IFG (e.g., $tmp\_t \blacktriangleright_{<1802,1817>} login\_d$) can represent multiples flows (e.g., three flows $tmp\_t \overset{T}{\triangleright}_{[1802,1804]} login\_d$, $tmp\_t \overset{T}{\triangleright}_{[1805,1809]} login\_d$ and $tmp\_t \overset{T}{\triangleright}_{[1815,1817]} login\_d$) using only two dates (e.g., 1802 and 1817).

Obviously, using multiple direct flows instead of single direct flows clearly provides an over-approximation of the latter ones on the system. But this has the great advantage of highly reducing the IFG's size. The number of nodes is theoretically bounded by $n = \mathcal{O}(|\mathcal{SC}|)$, whereas the number of vertexes is theoretically bounded by $v = \mathcal{O}(|\mathcal{SC}| \times |\mathcal{SSC}| - 1)$. It thus can fit in memory[2].

[2]E.g., for a Gentoo Linux OS, $n < 580$ and $v < 80000$.

Within IFG, the search of an indirect flow between two security contexts is done by searching for a path between the two nodes corresponding to the security contexts. We are using a Breadth First Search (BFS) algorithm that also verifies causal dependency between each direct flows that compose the indirect one. The use of a BFS algorithm allows to have a theoretically bounded complexity of searching indirect flow due to the nature of IFG.

When we want to compare indirect flows, we need to find all the occurrence of those flows. Our indirect flow algorithm is able to enumerate all the flows between two security context. For example, on the Figure 4, the algorithm enumerates two flows from sc_1 to sc_5. Those two flows have different ending date: 1010 for the first one f_{3a} ($\#1 \rightarrow \#2 \rightarrow \#4$) and 1150 for the second one f_{3b} ($\#1 \rightarrow \#3 \rightarrow \#5$). This ability is important for the race condition algorithm as we search for temporal relation between flows. If we have only one of the two occurrences of the flow, we can miss a race condition. For example, if we have only f_{3a} that ends at 1010 and we want to compare it with a flow f_2 that start at 1025, the condition $s_2 \leq e_3$ will be false. But if we have the two flows f_{3a} and f_{3b}, the condition will be true as $start(f_2) \leq end(f_{3b})$.

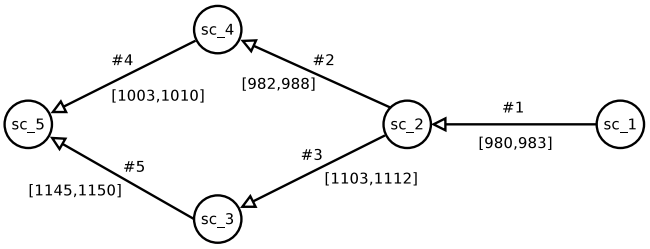

Fig. 4. Search all the occurrences of a flow between sc_1 and sc_5 in the IFG

Using the IFG and the general information flow algorithm, we are able to compile the property 3.1 into an algorithm. The algorithm 1 allows to detect any race condition between a legal entity lsc and a malicious one msc. This algorithm uses a function that returns every flows corresponding to three general flows between lsc and msc. Indeed, we need to verify the temporal relationships between those flows as described in the property 3.1 ($(s_1 \leq e_2) \wedge (s_2 \leq e_3)$). In the algorithm, it_1 is the current interaction i.e. the interaction that is in the authorization process. The algorithm 1 is a variation of the property 3.1 with the third flow between the shared object osc and the legal security context lsc must be a direct flow. In practice, it is usually the case. Moreover, this algorithm goes further than previous approaches as it takes into account indirect flows in two case out of three. This variation of the property 3.1 allows to reduce the overall complexity of the algorithm. Indeed, we only use the indirect flow algorithm two times instead of three.

F. Protecting an Operating System

This section presents PIGA-DYN-PROTECT i.e. our dynamic approach to guarantee the security properties expressed

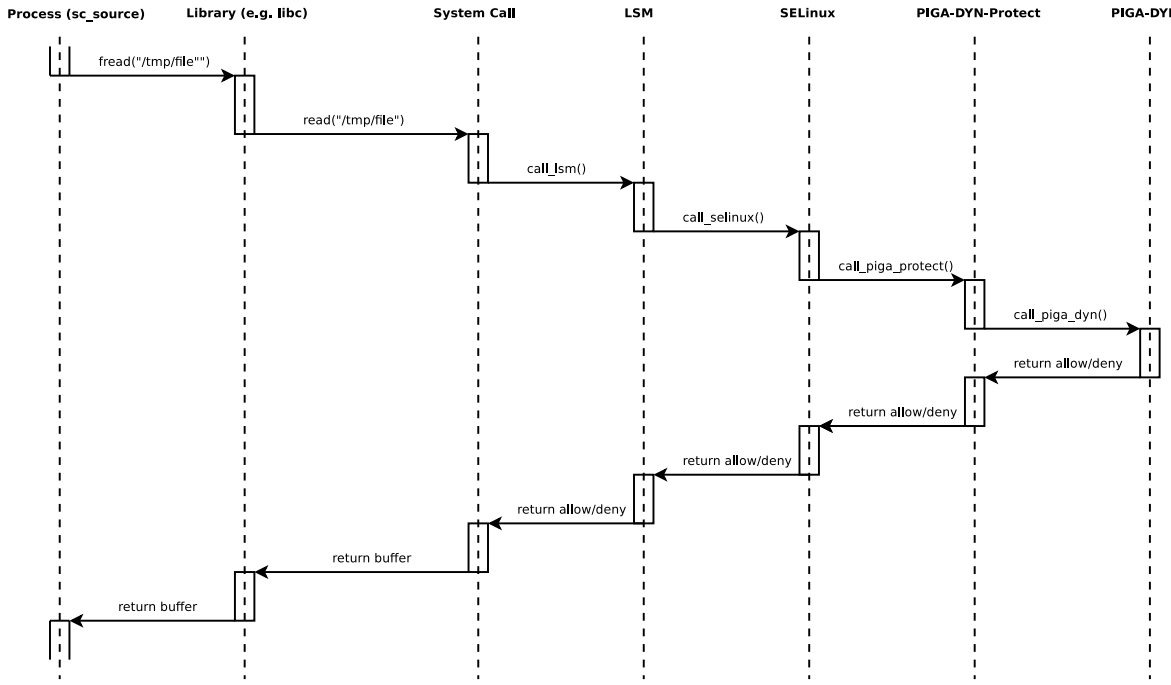

Fig. 5. Global architecture of our protection system

Algorithm 1 Algorithm to detect race condition from lcs to mcs : $No\_Race\_Condition(lsc, msc, it_1, IFG)$

Require: lsc, msc, it_1, IFG
  if $op(it_1) \in \mathcal{WEO}$ **or** $op(it_1) \in \mathcal{REO}$ **then**
    $iflow_1 = lcs \overset{T}{\rightarrowtail} dest(it_1)$
    $iflow_2 = mcs \overset{T}{\rightarrowtail} dest(it_1)$
    $list\_flow_1 = searchAllPath(iflow_1, IFG)$
    $list\_flow_2 = searchAllPath(iflow_2, IFG)$
    if $list\_flow_1 \neq$ NULL **and** $list\_flow_2 \neq$ NULL **then**
      for all $flow_1 \in list\_flow_1$ **do**
        for all $flow_2 \in list\_flow_2$ **do**
          if $start(flow_1) \leq end(flow_2)$ **then**
            return FALSE
          end if
        end for
      end for
    end if
  end if
  return TRUE

with our language. PIGA-DYN-PROTECT enforces a policy that is a set of security properties. Our architecture reuses SELinux security contexts. SELinux provides a context for all processes, users and system resources. In contrast with approaches like [30], [36], [37], [17], [16], we do not require a unique context for each entity e.g., $/tmp/file1$ and $/tmp/file2$ are grouped in a common label tmp_t. However, the SELinux policy is not required. Our solution is a satisfying approach since defining a consistent SELinux policy is a complex task. On the other hand, [38], [39] show that a SELinux protection policy can still present around a million of attack vectors. So, adding a protection against *RC* over SELinux security contexts is very efficient.

Figure 5 describes the global architecture of our protection. This architecture is composed of several components. We choose to illustrate each component based on the execution of a system call. In this example, an application calls a function `fread`. The library C (`libc`) provides this function. Then, the library calls the `read` system calls to communicate with the kernel of the operating system. This system call allows to execute a read in kernel space. Indeed, it is the only way to perform an input/output access to transfer information from or to a physical resources like a hard drive. Before this input/output operation, the kernel checks the discretionary permissions such as read, write and execution rights. If the operation is allowed by DAC then it calls the Linux Security Module (LSM). LSM is used to plug new security modules that hook system calls. LSM is used to call the SELinux module. Then, this module applies Type Enforcement on the system i.e. it sets a context for each entity of the operating system. PIGA-DYN-Protect is called by SELinux and collects all the information needed to build the trace of the system calls. This trace is sent to PIGA-DYN that uses it to build the IFG. It computes that graph to verify if the current system call goes against a security property. The decision is returned to PIGA-DYN-Protect then to SELinux. Finally, LSM receives the decision and allows or denies the system call.

In practice, for our *RC* property, the decision is taken at the last step of an attack attempt i.e. on the third system call. Thus, our solution prevents efficiently the third flow of a *RC* attack.

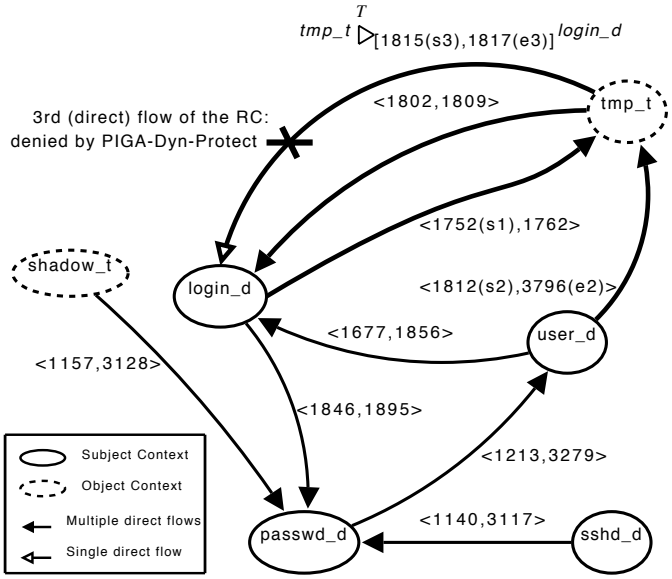

Fig. 6. IFG for the real protection against the login *RC* attack.

Moreover, our solution protects against unknown attacks (e.g., 0-Day attacks) and various covert channels associated with indirect flows.

G. Example of prevention of a Race Condition

Let us give an example of the No_Race_Condition property. The property: No_Race_Condition($login\_u$:$login\_r$:$login\_d$, $user\_u$:$user\_r$:$user\_d$, T) prevents a user process ($user\_u$:$user\_r$:$user\_d$) to interfere with the login process ($login\_u : login\_r : login\_d$) via a *RC* attack.

The IFG given in Figure 3 describes the so-called 'login *RC*' attack scenario. Thus, the first flow of the considered property can be seen as the direct flow $login\_d \overset{T}{\triangleright}_{[1752,1762]} tmp\_t$. The second flow is a direct flow $user\_d \overset{T}{\triangleright}_{[1812,3796]} tmp\_t$. The third flow is the direct flow $tmp\_t \overset{T}{\triangleright}_{[1802,1817]} user\_d$. Using this IFG, we are able to block the system call corresponding to the third flow at the kernel level. It thus avoids to exploit the login's vulnerability via a *RC* attack.

IV. Real World Experiment

To make real world experimentation of our solution, we have setup a high-interaction honeypot. It contains services with exploitable flaws to ease the attacks. A virtual machine provides the different services. During six month, we experimented many instances of the No_Race_Condition security property. Let us give the results for two of them. The first one was the protection against the Login *RC* attack (already mentioned) and the second was the protection against the PhpBB *RC* attack. Others instances, not presented here, cover attacks on filesystems, network services and shell scripts.

As any mandatory protection, PIGA-Dyn-Protect is an over-approximation of the attacks thus it can generate false positives. On the other side, the false negatives are due to a bad definition of the property or an incorrect configuration of it. Thus, it can be fixed through a new definition for the property or another configuration of it.

A. The Login attack RC to gain root privilege

To welcome attackers for the first evaluated attack: the 'Login Race Condition', we setup SSH servers on three machines of our honeypot. Those servers accepted attackers with the automatic creation of accounts when couples of login/password were tried.

The attack scenario is the following. An attacker ($user\_d$) connects to the SSH server (ssh_d). Then, the attacker uses the authentication service ($passwd\_d$) and gains a user session ($user\_d$) on the machine. He uses his session to execute the login command ($login\_d$). Then, he exploits the login's vulnerability by changing a value stored in the temporary file (tmp_t). When the login process comes back later to read the (modified) value, that allows a privilege escalation of $user\_d$ to $login\_d$ that has root privileges. The attacker uses this privilege escalation to modify the shadow passwords file $shadow\_t$ and especially the root password. The attacker then uses the login command again to open a root session ($root\_d$) with the new root password. This attack only involves direct information flows.

Let us consider a system without our protection solution in order to detail the 'Login *RC*' attack scenario. The IFG given in Figure 3 corresponds to the violation of a No_Race_Condition security property. In this figure, the temporal constraints (line #4) of the No_Race_Condition security property between $user\_d$ and $login\_d$ are true: $(1752(s_1) < 3796(e_2)) \wedge (1812(s_2) < 1817(e_3))$.

Our PIGA-Dyn-Protect module cancels the last interaction of the third flow in order to avoid the *RC* success. In the example, PIGA-Dyn-Protect denies the execution of the interaction at system date 1817. Thus PIGA-Dyn-Protect cancels the interaction corresponding to the (single) direct flow $tmp\_t \overset{T}{\triangleright}_{[1815,1817]} login\_d$: the third flow violating the No_RC property does not appear in the real IFG, as shown in Figure 6.

As shown in the figure 8, we were able to monitor multiple attacks during the six months experimentation. Moreover, through SELinux logs analysis [40], we were able to validate that all the attacks were detected by our solution. With the protection mode, all the attacks were blocked and no attacker was able to gain root privilege.

B. PhpBB RC for Remote Shell Execution

We also experimented another kind of *RC* attacks: the PhpBB *RC* attack that can lead to a 'Remote Shell Execution'. In order to collect attacks, we build a fake PhpBB forum. We also advertised about *cgi scripts* (bash and binary) execution.

For this experiment, the security property against *RC* was configured as the following: No_Race_Condition($apache\_u$:$apache\_r$:$apache\_d$, $apache\_u$:$apache\_r$:$phpbb\_d$, T). Compared to the login *RC*,

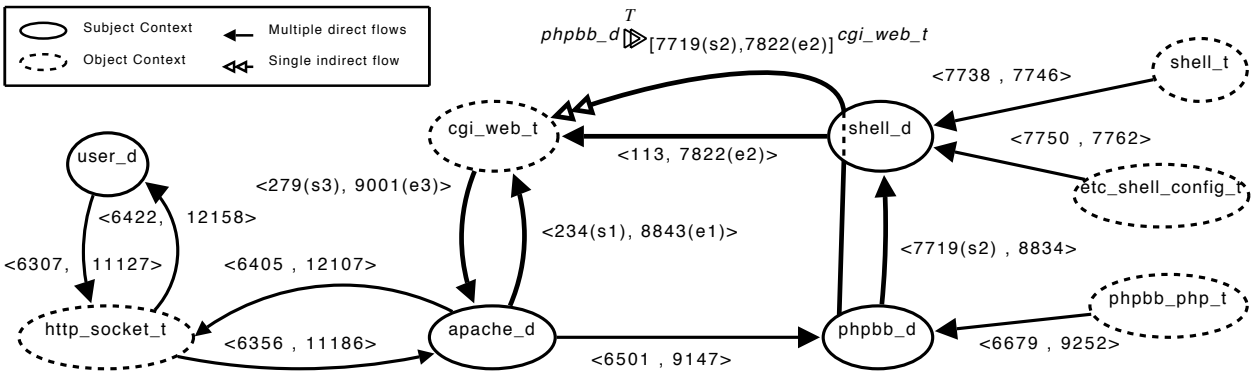

Fig. 7. IFG for the phpBB *RC* attack scenario.

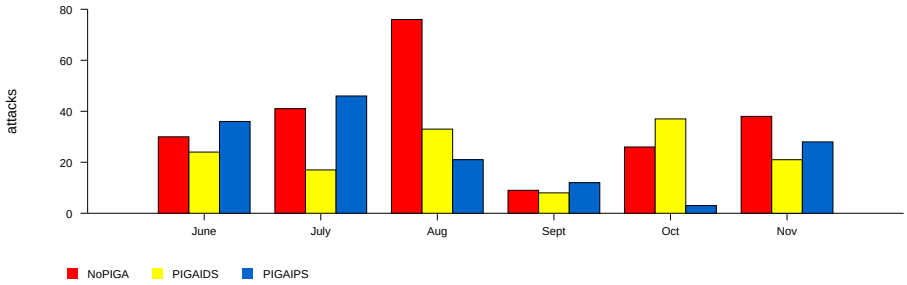

Fig. 8. Number of attacks per month per machine for the login race condition

this attack is a more complex one. The Figure 7 describes the PhpBB attack scenario. First, the attacker ($user\_d$) has to connect to the apache server ($apache\_d$) through a socket ($http\_socket\_t$) and accesses to the PhpBB forum ($phpbb\_d$). Then, the attacker uses the remote execution exploit in the PhpBB forum to execute a shell ($shell\_d$). He uses the shell to modify the cgi scripts (cgi_web_t). In summary, this *RC* based PhpBB attack involves direct and indirect information flows. The first flow of the attack is $apache\_d \overset{T}{\triangleright}_{[234,8843]} cgi\_web\_t$. The second one is indeed an indirect flow: $phpbb\_d \overset{T}{\blacktriangleright}_{[7719,7822]} cgi\_web\_t$ (via $shell\_d$). The third flow is a direct one: $cgi\_web\_t \overset{T}{\triangleright}_{[279,9001]} apache\_d$.

In the *RC* PhpBB attack scenario given in Figure 7, the temporal relations between flows violate the temporal constraints (line #4) of the No_Race_Condition security property between $apache\_d$ and $phpbb\_d$: $(234(s_1) < 7822(e_2)) \wedge (7719(s_2) < 9001(e_3))$.

Again, our PIGA-DYN-PROTECT module denies the last interaction of the third flow. It thus forbids the execution of the corresponding interaction at the system date 9001.

As shown in the figure 9, we were able to monitor multiple attacks during the six months period of our experimentation. Same as login RC attacks, all the phpBB attacks were detected and blocked.

V. EFFICIENCY

A. Completeness

To evaluate the correctness of our approach, we configured our honeypot hosts with PIGA-DYN-PROTECT in both detection and prevention mode. That way, we could verify that every detected attack was prevented or not. During the six months of experiment, we detected 146 instances of the login *RC* attack and 574 instances of the PhpBB *RC* attack. All attacks were blocked by our protection mechanism.

B. Performances

In order to evaluate the performances of our solution, several benchmarks are proposed for three different configurations:

- a classical Linux system with DAC and SELinux TE.
- a Linux system with DAC and SELinux TE with PIGA-DYN in analysis mode for detecting the violation of the required security properties (PIGA-IDS).
- a Linux system with DAC and SELinux TE with PIGA-DYN in protection mode for enforcing the required security properties (PIGA-IPS).

The hardware configuration was the same, a Pentium-4 3Ghz with 1Gb of memory.

We use lmbench [41] suite running on these three machines to measure bandwidth and latency. Lmbench attempts to measure performance bottlenecks in a wide range of system applications. These bottlenecks have been identified, isolated

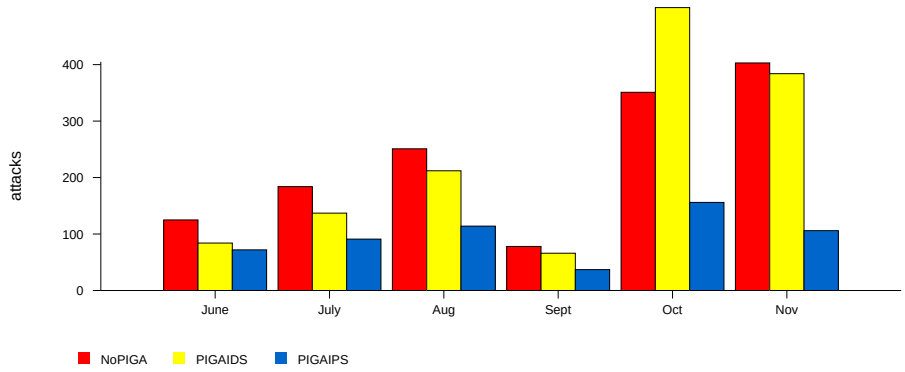

Fig. 9. Number of attacks per month per machine

Operation	Description
memory read	Measuring time to read x byte word from memory
memory write	Measuring time to write x byte word to memory
memory read/write	Measuring time to read an write x byte word to memory

TABLE I
MEMORY OPERATIONS FROM LMBENCH

and reproduced in a set of microbenchmarks that measures the system latency and bandwidth of data movement. First, we focus on the memory subsystem and measure bandwidth with various memory operation listed in the table I.

As shown in the Figure 10, the difference between the three different configurations is small. With data blocks larger than 512Kb, the three configurations have almost the same performances. With data blocks smaller than 512Kb, the overhead due to our security component is unnoticeable. In the worst case, like memory read/write, the overhead is 5%. Consequently, we can state that our security component has little to no influence on data copy to and from the memory.

In a second time, we used lmbench to measure latency in five different parts of the operating system:

- Process: it creates four different types of process and evaluates the time it takes 1) to invoke a procedure, fork a process and close it, 2) fork a process and invoke the execve system call and 3) fork a process and invoke an interactive shell.
- Context switching: it measures the context switching time for a number of processes of a given size (in Kb). The processes are all connected through a ring of Unix Pipe where each process reads its input pipe, does some work and writes in its output pipe i.e. the input pipe of the next process.
- System call: it measures the time to write one byte to $/dev/null$. It permits to evaluate the interaction time with the system.
- Filesystem: it measures the time to create and delete files with a size between 0 and 10Kb.
- Network: it measures the time taken to make a HTTP request ($GET/$) on a LAN and a WAN HTTP server.

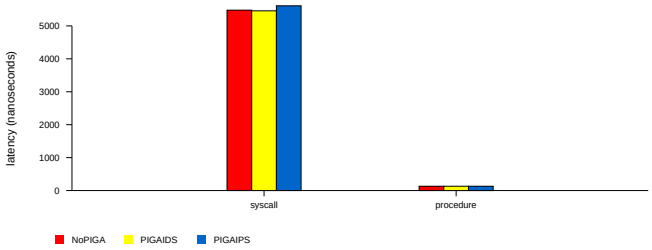

Fig. 11. Syscall and Procedure Latency for the three operating systems

Figure 11 displays, on the first set of columns, the latency (in nanoseconds) to invoke a system call. The operating system without our solution and with our solution in analysis mode have the same latency. The overhead due to our solution in protection mode is minimal (2% on average). Thus, our solution has little to no influence on the system call latency. The second set of columns shows the latency when invoking a function in a program. The three operating systems have the same latency. Our solution does not change the procedure call latency. Indeed, the call of a procedure inside a program does not require to compute any authorization.

Figure 12 displays the average time (in microseconds) to create and destroy three types of process: first one is just a creation and a deletion, the second one is a creation then an execution (execve), the third one is a creation then the invocation of a shell (/bin/sh). The overhead of our two solutions is high. Indeed, thousands of fork system calls, within few seconds, create thousands of path searches. The overhead is about 300% when using our solution in analysis mode and

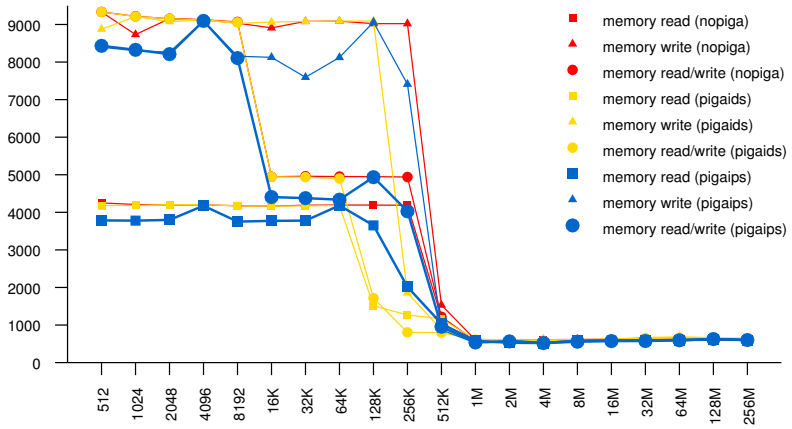

Fig. 10. Memory bandwidth performance

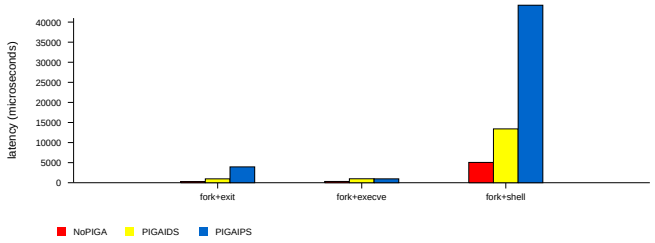

Fig. 12. Fork Latency for the three operating systems

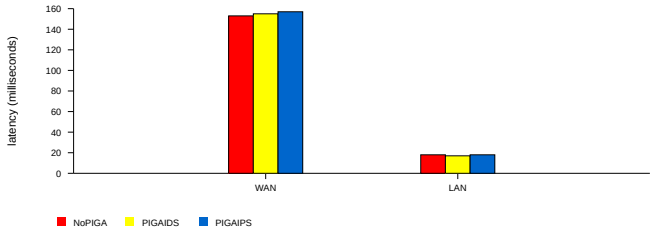

Fig. 13. Network Latency for the three operating systems

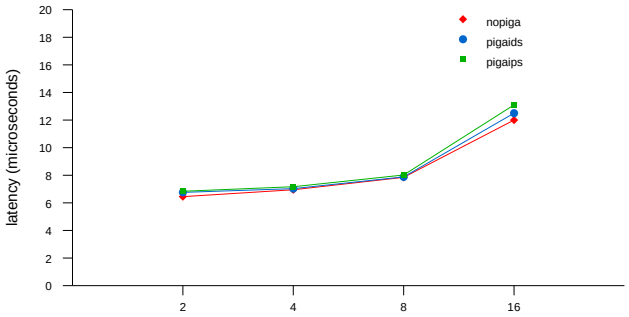

Fig. 14. Context Switch Latency for the three operating systems

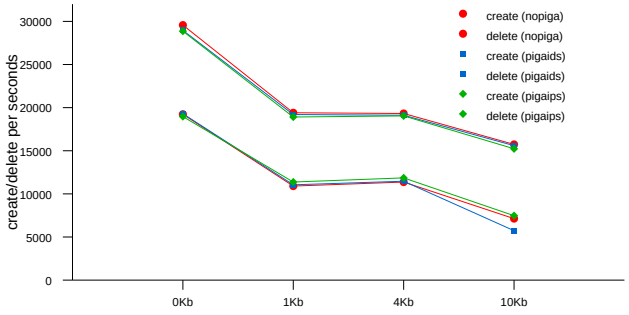

Fig. 15. Filesystem Latency for the three operating systems

800% in protection mode. This overhead can be reduced by pre-computing the path on the IFG, such approaches are under development. However, thousands of forks within a limited time really is unusual. So, this overhead is not relevant of a common usage.

As one can see on the Figure 13, the network latency is the same for the three operating systems. The amount of access control verifications for a network system call is small since our solution does not have to do expensive path searches.

Figure 14 shows the average latency when switching from a process to another one when 2, 4, 8 and 16 processes communicate together through a ring of Unix pipe. The overhead in analysis mode is about 4% on average. A context switching generates dozens of access control requests. Our solution is efficient since it computes the dozens of requests in a very short delay. With our solution in protection mode, the overhead is less than 6%.

Figure 15 shows the timing of creating and deleting files on a filesystem (an ext3 filesystem). It shows how many files can be created in one second with a size of 0, 1, 4 or 10Kb and

how many files can be deleted in one second with the same size. The overhead of our two solutions is less than 1%.

This performance analysis shows that our solution (in analysis or protection mode) is efficient for common usages. The fork overhead corresponds to an unusual stress of the system. To cope with such situations, we are looking at different precomputing and optimization methods while keeping a stable memory usage.

Our paper presents a novel approach to protect a complete operating system against attacks using Race Conditions. It provides a large state of art showing that this problem still is opened. Our solution prevents efficiently against both direct race conditions and indirect ones. A new protection property is defined to cope with these various race conditions. That property fits with the enforcement by an operating system against RCs.

An implementation is proposed for guaranteeing the proposed security property within a Linux system. Our implementation provides a dedicated MAC approach. For compatibility reasons, SELinux contexts are reused. However, the SELinux enforcement is not required and the solution supports ordinary DAC systems. It requires only security contexts associated to the different system resources.

Our approach computes the illegal activities dynamically. It allows to easily define security properties against RCs attacks.

Real examples show the efficiency of our approach for preventing real attacks using race conditions. A complete benchmark has been carried out to show the low overhead of our solution. Improvements are on the way to cope with unusual stress of the system.

Since our approach allows to formalize a wide range of security properties, future works will tackle other security properties dealing with integrity, confidentiality and availability.

REFERENCES

[1] J. Rouzaud Cornabas, P. Clemente, and C. Toinard, "An Information Flow Approach for Preventing Race Conditions: Dynamic Protection of the Linux OS," in *Fourth International Conference on Emerging Security Information, Systems and Technologies SECURWARE'10*, (Venise Italie), pp. 11–16, 07 2010.

[2] R. H. B. Netzer and B. P. Miller, "What are race conditions? some issues and formalizations," *LOPLAS*, vol. 1, no. 1, pp. 74–88, 1992.

[3] Y. Yu, T. Rodeheffer, and W. Chen, "Racetrack: efficient detection of data race conditions via adaptive tracking," *SIGOPS Oper. Syst. Rev.*, vol. 39, no. 5, pp. 221–234, 2005.

[4] T. Tahara, K. Gondow, and S. Ohsuga, "Dracula: Detector of data races in signals handlers," (Beijing, China), pp. 17–24, IEEE Computer Society, 2008.

[5] W. S. McPhee, "Operating system integrity in os/vs2," *IBM Syst. J.*, vol. 13, no. 3, pp. 230–252, 1974.

[6] M. Bishop, "Race conditions, files, and security flaws: or, the tortoise and the hare redux," *Technical Report*, vol. 95, sept. 1995.

[7] M. Bishop and M. Dilger, "Checking for race conditions in file accesses," *Computing Systems*, vol. 9, pp. 131–152, 1996.

[8] R. H. Netzer and B. P. Miller, "On the complexity of event ordering for shared-memory parallel program executions," in *In Proceedings of the 1990 International Conference on Parallel Processing*, pp. 93–97, 1990.

[9] H. Chen and D. Wagner, "Mops: an infrastructure for examining security properties of software," in *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, (New York, NY, USA), pp. 235–244, ACM, 2002.

[10] B. Chess, "Improving computer security using extended static checking," in *Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on*, pp. 160 – 173, 2002.

[11] B. Schwarz, H. Chen, D. Wagner, J. Lin, W. Tu, G. Morrison, and J. West, "Model checking an entire linux distribution for security violations," in *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference*, (Washington, DC, USA), pp. 13–22, IEEE Computer Society, 2005.

[12] C. Ko and T. Redmond, "Noninterference and intrusion detection," in *Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on*, (Oakland, CA, USA), pp. 177–187, 2002.

[13] K.-s. Lhee and S. J. Chapin, "Detection of file-based race conditions," *International Journal of Information Security*, vol. 4, pp. 105–119, 2005. 10.1007/s10207-004-0068-2.

[14] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen, "Detecting past and present intrusions through vulnerability-specific predicates," in *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, (New York, NY, USA), pp. 91–104, ACM, 2005.

[15] C. Cowan, S. Beattie, C. Wright, and G. Kroah-Hartman, "Raceguard: kernel protection from temporary file race vulnerabilities," in *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, (Berkeley, CA, USA), pp. 13–13, USENIX Association, 2001.

[16] E. Tsyrklevich and B. Yee, "Dynamic detection and prevention of race conditions in file accesses," in *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, (Berkeley, CA, USA), pp. 17–17, USENIX Association, 2003.

[17] P. Uppuluri, U. Joshi, and A. Ray, "Preventing race condition attacks on file-systems," in *Proceedings of the 2005 ACM symposium on Applied computing - SAC '05*, (New York, New York, USA), p. 346, ACM Press, 2005.

[18] F. Schmuck and J. Wylie, "Experience with transactions in quicksilver," in *SOSP '91: Proceedings of the thirteenth ACM symposium on Operating systems principles*, (New York, NY, USA), pp. 239–253, ACM, 1991.

[19] C. P. Wright, R. Spillane, G. Sivathanu, and E. Zadok, "Extending acid semantics to the file system," *Trans. Storage*, vol. 3, no. 2, p. 4, 2007.

[20] D. Mazieres and M. Kaashoek, "Secure applications need flexible operating systems," in *Operating Systems, 1997., The Sixth Workshop on Hot Topics in*, pp. 56 –61, 5-6 1997.

[21] D. Tsafrir, T. Hertz, D. Wagner, and D. Da Silva, "Portably solving file tocttou races with hardness amplification," in *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, (Berkeley, CA, USA), pp. 1–18, USENIX Association, 2008.

[22] D. F. Ferraiolo and D. R. Kuhn, "Role-based access controls," in *15th National Computer Security Conference*, (Baltimore, MD, USA), pp. 554–563, Oct. 1992.

[23] ITSEC, "Information Technology Security Evaluation Criteria (ITSEC) v1.2," technical report, June 1991.

[24] P. A. Loscocco, S. D. Smalley, P. A. Muckelbauer, R. C. Taylor, S. J. Turner, and J. F. Farrell, "The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments," in *Proceedings of the 21st National Information Systems Security Conference*, (Arlington, Virginia, USA), pp. 303–314, Oct. 1998.

[25] J. Anderson, "Computer security threat monitoring and surveillance," tech. rep., James P. Anderson Company, Fort Washington, Pennsylvania, April 1980.

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

[27] T. Fraser and L. Badger, "Ensuring continuity during dynamic security policy reconfiguration in dte," pp. 15 –26, may. 1998.

[28] N. Li, Z. Mao, and H. Chen, "Usable mandatory integrity protection for operating systems," in *SP '07: Proceedings of the 2007 IEEE Symposium on Security and Privacy*, (Washington, DC, USA), pp. 164–178, IEEE Computer Society, 2007.

[29] T. Fraser, "LOMAC: Low Water-Mark integrity protection for COTS environments," *Proceeding 2000 IEEE Symposium on Security and Privacy. S&P 2000*, pp. 230–245, 2000.

[30] Z. Mao, N. Li, H. Chen, and X. Jiang, "Trojan horse resistant discretionary access control," in *SACMAT '09: Proceedings of the 14th ACM symposium on Access control models and technologies*, (New York, NY, USA), pp. 237–246, ACM, 2009.

[31] S. Vandebogart, P. Efstathopoulos, E. Kohler, M. Krohn, C. Frey, D. Ziegler, F. Kaashoek, R. Morris, and D. Mazières, "Labels and event

processes in the asbestos operating system," *ACM Trans. Comput. Syst.*, vol. 25, no. 4, p. 11, 2007.

[32] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières, "Making information flow explicit in histar," in *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, (Berkeley, CA, USA), pp. 19–19, USENIX Association, 2006.

[33] P. Efstathopoulos and E. Kohler, "Manageable fine-grained information flow," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 4, pp. 301–313, 2008.

[34] M. Krohn and E. Tromer, "Noninterference for a practical difc-based operating system," in *SP '09: Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, (Washington, DC, USA), pp. 61–76, IEEE Computer Society, 2009.

[35] P. Clemente, J. Rouzaud-Cornabas, and C. Toinard, "From a generic framework for expressing integrity properties to a dynamic enforcement for operating systems," in *Transactions on Computational Science XI* (M. Gavrilova, C. Tan, and E. Moreno, eds.), vol. 6480 of *Lecture Notes in Computer Science*, pp. 131–161, Springer Berlin / Heidelberg, 2010.

[36] H. Liang and Y. Sun, "Enforcing mandatory integrity protection in operating system," in *ICCNMC '01: Proceedings of the 2001 International Conference on Computer Networks and Mobile Computing (ICCNMC'01)*, (Washington, DC, USA), p. 435, IEEE Computer Society, 2001.

[37] N. Li, Z. Mao, and H. Chen, "Usable mandatory integrity protection for operating systems," in *Security and Privacy, 2007. SP '07. IEEE Symposium on*, (Oakland, CA, USA), pp. 164–178, May 2007.

[38] J. Briffaut, J.-F. Lalande, and C. Toinard, "A proposal for securing a large-scale high-interaction honeypot," in *Workshop on Security and High Performance Computing Systems* (R. K. Guha and L. Spalazzi, eds.), (Cyprus), ECMS, 2008.

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

[40] M. Blanc, P. Clemente, J. Rouzaud-Cornabas, and C. Toinard, "Classification of malicious distributed selinux activities," *Journal Of Computers*, vol. 4, pp. 423–432, may 2009.

[41] L. McVoy and C. Staelin, "lmbench: portable tools for performance analysis," in *ATEC '96: Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, (Berkeley, CA, USA), pp. 23–23, USENIX Association, 1996.