# Compacting Security Signatures for PIGA IDS

Pascal Berthomé, Jérémy Briffaut, Pierre Clairet
ENSI de Bourges – Université Orléans – LIFO 18020 Bourges Cedex, France
Email: {pascal.berthome,jeremy.briffaut,pierre.clairet}@ensi-bourges.fr

***Abstract*—PIGA (Policy Interaction Graph Analysis) is a tool that detects malicious process behaviours by analysing the operating system activities. This tool uses signatures that represent illegal activities of some malicious user. These signatures are generated from a graph that models the performed operations at operating system (OS) level. For usual security properties, the number of signatures is large and they are stored in the memory during the detection process. In this paper, we present a way to reduce the memory required to store the signatures while preserving their quality. The methodology is derived from the modular decomposition of graphs. We investigate the impact of such an approach for the confidentiality property. The efficiency of the methodology is evaluated on interaction graphs of real operating systems. The number of signatures is divided by 20 for the tested confidentiality property.**

***Keywords-security; operating system; modular decomposition.***

## I. Introduction

Security in the Information Technology domain relies on global properties that need to be ensured. Integrity and confidentiality are two of the most important ones. Many others have been defined in the literature but they are mainly based on the noninterference principle [1][2]. PIGA [3] is a tool that can be used as an intrusion detection system (IDS) or an intrusion prevention system (IPS). An intrusion is an action that breaks a security property that is declared in the security policy.

In order to prevent illegal activities, PIGA [3] monitors the kernel activities (system calls) and examines the traces of the different processes. In [4], the authors present an overview of the different techniques of intrusion detection based on externally specified rules. For example, Remus [5] or BlueBoX [4] have a policy-based approach. The goals of these examples are different. The aim of Remus is to minimise the performance impact whereas BlueBoX tries to minimise the impact on the kernel. However, these systems detect some intrusion without taking into account the previous operations performed in the OS. PIGA defines on the contrary, signatures that describe the potential attacks on the operating system, by considering the temporal order of the operations.

One important feature of PIGA is to detect security property breaks by using a set of signatures. Many intrusion detection system are signature-based. For example, the Argos System also computes signatures; however, these signatures are generated from the examples of attacks that have already occurred [6]. The particularity of PIGA is that these signatures are generated off-line directly from the expression of the security properties. The drawback of PIGA is that the signature base is large for real systems and requires a significant memory space allocation for the detection process; furthermore, this base has to be restricted to small signatures because the computation requires a huge amount of memory, i.e., exponential with the length of the signatures [3]. For a complete system based on Fedora using graphical user interface the signatures base takes more than 500 MB.

This paper proposes an improvement of the PIGA system. In order to reduce the memory space needed, we propose a way to compact the information required for the intrusion detection. This paper focusses on the confidentiality property since it is the property that generates the largest number of signatures [3]. PIGA uses a graph to represent the system and its interactions, thus, this improvement is based on the reduction of this graph. For this, we exploit the modularity property of the input graph, i.e., the fact that some groups of nodes have the same behaviour considering the remaining part of the graph.

In order to present this contribution, the paper is organised as follows. In Section II, a brief presentation of PIGA is given and some security definitions are recalled. Section III presents the problem tackled in this paper. Section IV presents the major contribution of this paper, i.e., the use of the modular decomposition of graph to lower the size of the memory required to store signatures. It also provides a theoretical evaluation of the efficiency of our method. Section V shows that our strategy can be used to decrease the size of the information stored for detecting confidentiality threat on real systems. Section VI draws some conclusions and the direction of further works.

## II. General Context

This section summarises the study provided for the PIGA system in [3]. The goal of PIGA is to monitor all the system activities and detect those that break rules defined by the administrator. This section proposes to recall the definitions of the main terms of security used in this paper through the prism of the PIGA system.

We present the process for generating a set of signatures from the security policy and the interaction graph. This process is explained throughout the following subsections. In Section II-A, the security properties are presented. In Section II-B, we describe the fundamental elements of a system, i.e., the security contexts. In Section II-C, we recall how these contexts are connected through the interactions of the system. In Section II-D, we glue these elements together

 ISBN: 978-1-61208-209-7

in order to represent the system by a directed graph. Finally in Section II-E, we present how the signatures are computed for the confidentiality property.

### A. System and security properties

A system can be seen as a state machine, where a state is characterised by the state of all the available resources [1]. The state of the system changes at each system call. In order to ensure the security of the system, the administrator defines the rules that represent the frontier between the *safe* and the *unsafe* state of the system. These rules are called security properties and the set of these security properties is called security policy. They are mainly based on the noninterference principle [7].

*Integrity* and *confidentiality* are the most common security properties [8]. The aim of the *confidentiality* property is to ensure there is no illegal information transfer. For example, a *confidentiality* property of the file `/etc/shadow` for the users prevents any information transfer from `/etc/shadow` to any user. The goal of the *integrity* property is to ensure there is no unwanted modification. For example, an *integrity* property of `/etc/shadow` for the users prevents any modification of `/etc/shadow` by a user.

### B. Security contexts

PIGA is implemented upon SELinux, but it could be defined upon other mandatory access control (MAC) systems. Thus, it uses SELinux security contexts to label the resources of the system (file, process, resource, network port, ...). By extension, we say that two resources belong to the same security context if they have the same label.

In their simplest format, the security contexts in SELinux are defined by three elements: the user, the role and the type of the entity. For example, the security context of `/etc/shadow` is `system_u:object_r:shadow_t`. Several entities playing the same role can have the same security context. For example, `user_u:object_r:user_home_t` contains all the files in the home directory of the user.

### C. Interactions and sequences

Any elementary system operation involves two resources, thus implying two security contexts [1]. This defines an *interaction* between two contexts. In SELinux, only allowed interactions are described explicitly. However, as shown in [3], using these allowed interactions cannot ensure that the system would remain in a safe state. Some security properties can be broken by performing successive operations. A *sequence* is then defined as a set of successive interactions, e.g., an information flow between two security contexts.

The aim of the PIGA system is to automate the generation of the sequences that break the security properties, defining the signatures as explained in the following sections.

### D. Interaction and flow graphs

As shown before, PIGA models the system with a directed graph that represents the security contexts and the set of allowed interactions. More precisely, the system is represented by the interaction graph $G = (V, A)$, such that:

- $V$ represents the security contexts, and
- $A$ the set of interactions between those contexts.

Furthermore, any arc in this graph is labelled. The label of the arc from the security context $sc_1$ to the security context $sc_2$ represents the set of operations from $sc_1$ to $sc_2$ allowed by the MAC policy of the system. For example,

```
user_u:user_r:xserver_t
    system_u:object_r:mtrr_device_t
            file  { write read };
```

represents the possible interactions from the context `user_u:user_r:xserver_t` to `system_u:object_r:mtrr_device_t`. It means that any element having `user_u:user_r:xserver_t` security context can perform read and write operations on the files of the context `system_u:object_r:mtrr_device_t`. SELinux provides a wide range of permissions that can be allowed between different contexts. Figure 1 shows a simplified interaction graph.

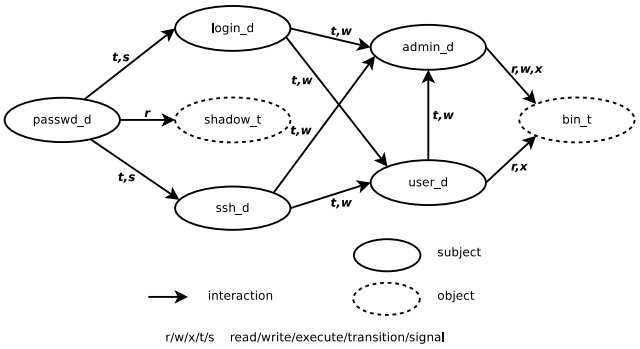


Fig. 1. A simplified interaction graph

From the interaction graph, we derive other graphs that are used in the generation of signatures. These new graphs are obtained by filtering the labelling function. These filtering operations may have different consequences on the arcs: an arc can be removed from the graph if the resulting label is empty; an arc also can change its direction by considering any property to be preserved with this operation. In the case of the confidentiality property, it uses the flow graph $FG = (V, A)$ where $V$ is the set of the security contexts and $A$ represent all the possible information transfers between the security contexts. The flow graph can be deducted from the interaction graph by removing the labels that do not imply flow transfer, maybe removing some arcs at this step. The remaining arcs deal with `read` and `write` operations. If a write operation occurs from context $sc_1$ to $sc_2$, there is a flow transfer from $sc_1$ to $sc_2$ and leads to an arc in the flow graph from $sc_1$ to $sc_2$. If a read operation occurs from context $sc_1$ to $sc_2$, then there is a flow transfer from $sc_2$ to $sc_1$ and leads to an arc in the flow graph from $sc_2$ to $sc_1$. Figure 2 presents the resulting flow graph derived from the interaction graph of Figure 1.

### E. Signatures

A signature represents a set of actions, ordered or not, that breaks a security property. Note that a signature is derived

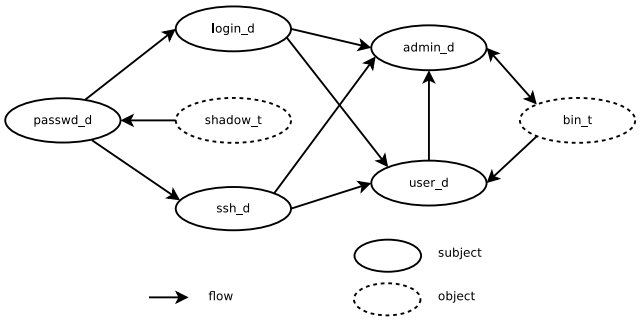


Fig. 2. The corresponding flow graph of the interaction graph of Figure 1

from a security property, but a security property generates several signatures. In PIGA, a signature represents an interaction, a sequence or a combination of sequences and/or interactions.

This paper focusses on confidentiality. The experiments of [3] show that this property generates the greatest number of signatures. In this case, a signature is a sequence that represents a succession of possible information transfers. For two contexts $sc_1$ and $sc_2$, $sign(sc_1, sc_2)$ represents the set of signatures between $sc_1$ and $sc_2$ and can be computed as the set of simple paths between $sc_1$ and $sc_2$ in the flow graph.

For example, from a system corresponding to Figure 1, the administrator would like to ensure that any file in the `shadow_t` security context cannot be accessible to any user (having security context `user_d`). Considering the flow graph in Figure 2, this would be possible that information from `shadow_t` arrives to `user_d` in only three steps without breaking the SELinux policy. The analysis of the flow graph results in 4 ways of breaking this security property :

```
shadow_t-> passwd_d-> ssh_d-> user_d
shadow_t-> passwd_d-> login_d-> user_d
shadow_t-> passwd_d-> ssh_d-> admin_d-> bin_t-> user_d
shadow_t-> passwd_d-> login_d-> admin_d-> bin_t-> user_d
```

When the PIGA system is active, all the interactions are audited and the system memorises the progression of all the signatures. If an interaction completes a signature, the system will forbid the last operation if it is in prevention mode and it will warn the user in permissive mode. Indeed, PIGA can be used either as an IDS or as an IPS.

## III. Problem definition

As seen in the previous section, PIGA computes all the signatures off-line and stores them in the system. As shown in [3], the number of signatures may be large even for systems having a small number of services. These signatures are required for the detection process. However, they take a huge place (several hundreds of megabytes) in the memory. Furthermore, the interaction graph may be so large that it is unrealistic to compute all the signatures. The problem comes first from memory excess, since the administrator of the PIGA system is not really time restricted to compute the signatures. The solution chosen in this case is to bound the length of the signatures. This approximation is acceptable for some systems, since the probability of defining an effective attack from a signature become more and more smaller with its length.

It is thus a challenge to find a good way to compress the signatures in order to lower the size of the storage required by the signatures. In the PIGA system, a simple compression is performed: the signatures are stored using a tree. After this compression, the size of the input signatures remains large.

Another challenge is to obtain a compression not impacting the efficiency of the detection process. For example, if the system only stores the pairs (source, target), the detection would be preponderant and slow down the system significantly.

This paper tackles the first problem of finding a generic way to compress the number of signatures. An important point is to ensure that any signature generated by the original policy is included in the set of compressed signatures. We will see that the reverse property is not required (any compressed signature only represents valid signatures). This comes from the specificity of the objects manipulated in this context. However, this paper tries to find the compressed set of signatures that minimises the number of unrealistic signatures in a compressed signature. Finally, the solution presented in this paper follows the philosophy of PIGA: the generation of a set of signatures using a graph that represents the system.

Since the signatures are simple paths in the flow graph, reducing the size of this graph should reduce the number of the signatures. In order to reduce the size of this graph, the idea is to group the nodes having the same behaviour into a single *meta-node*. For example, a simple analysis on the flow graph in Figure 2 shows that both contexts `ssh_d` and `login_d` have the same behaviour considering the other contexts. The idea developed in the remaining of this paper is to consider these two contexts as a single entity called `module`. Assuming this in this example, only two signatures should be considered.

```
shadow_t-> passwd_d-> module-> user_d
shadow_t-> passwd_d-> module-> admin_d-> bin_t-> user_d
```

This kind of property for a graph is known as the modular decomposition. To obtain an interesting compression, the input interaction graph must contain a significant number of modules. The modules should not be too large in order to maintain the detection process efficient. The goal of this paper is to evaluate the compression ratio of the application of the modular decomposition to realistic flow graphs. The following section presents the theoretical base of our technique.

## IV. Simplification of the input graph

The reader may refer to [9] for advanced concepts in graph theory. In this section, we first recall the definition of the modular decomposition on general undirected graphs in Section IV-A. Since the input graphs are directed, we show how to deal with this problem in Section IV-B. Section IV-C presents the general methodology for compressing signatures. In the original algorithm for computing the signatures, the signatures are defined as the set of the paths between the source and target security contexts. However, the decomposition process compacts some nodes into some *meta-nodes*, called *modules*. Thus, some source or target security contexts may disappear

 ISBN: 978-1-61208-209-7 

in this process. Section IV-D describes the process to make these contexts reappear. Finally, in Section IV-E, we provide an evaluation of the reduction of the number of signatures using this technique.

### A. Modular decomposition

The modular decomposition of a graph is a useful tool to represent graphs in a compact way by grouping nodes that have the same behaviour. Theoretical and algorithmic aspects of this decomposition are explored in [10]. This decomposition has several applications. For example, it has been used in graph drawing, where a module can be seen as a single node and indeed simplify the drawing [11]. This decomposition has been used in biology to simplify the protein-protein interaction network and obtain comprehensive view of this network [12]. All these examples show that the modular decomposition is used to simplify the input graph and then lower the complexity of solving initial the problem. In this paper, we use this type of graph decomposition to lower the size of the flow graph on which the signatures are computed.

*Definition 1:* Let $G = (V, E)$ be an undirected graph. A **module** $M$ of $G$ is a subset of $V$ such that:

$$\forall x \in V \setminus M \begin{cases} \forall y \in M, (x, y) \in E \text{ or} \\ \forall y \in M, (x, y) \notin E \end{cases}$$

Let $G_M$ be the subgraph of $G$ such that $G_M = (M, E')$ such that: $E' = \{(x, y) \in E \mid x, y \in M\}$. As shown in [10], module $M$ can have one of the three following types, depending on the connectivity of $G_M$:

- $M$ is series if $\overline{G_M}$ is not connected;
- $M$ is parallel if $G_M$ is not connected;
- $M$ is prime otherwise.

A module $M$ is **strong** if for any other module $M'$, either one is included in the other ($M \subseteq M'$ or $M' \subseteq M$) or they are totally disjoint ($M \cap M' = \emptyset$). The set of strong modules can be organised in a tree, called the **modular decomposition tree**, giving some hierarchical relationship between the modules. The root of this tree is the trivial module $V$ and the leaves are the other trivial modules: the single nodes. The modular decomposition tree of $G$ is designated by $MDTree(G)$.

Figure 3 represents $H$ the symmetrised version of the flow graph in Figure 2, where every arc between two nodes has been replaced by an edge between these nodes. The application of modular decomposition on $H$ generates the modular decomposition tree $MDTree(H)$ represented on Figure 4.

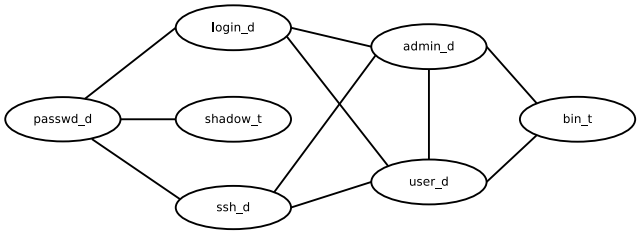


Fig. 3. The flow graph after symmetrization $H = (V, E)$

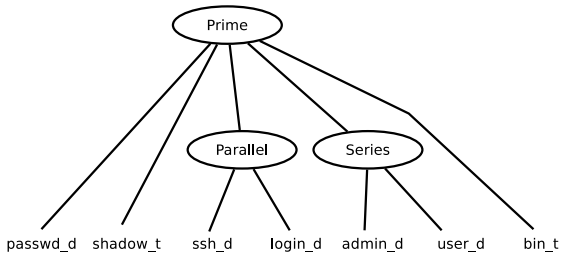


Fig. 4. The modular decomposition tree associated to $H$, $MDTree(H)$

A **quotient graph** can be obtained by grouping the nodes contained in the same module into a single node in the graph. The quotient graph of $G$ is called $Quot(G)$, e.g., the graph $Quot(H)$ represented on Figure 5 is the quotient graph of $H$. This operation makes the input graph smaller and thus easier to understand. In the following definition, we extend the notion of quotient graph to any partition of the vertices. This latter definition is also valid for directed graphs.

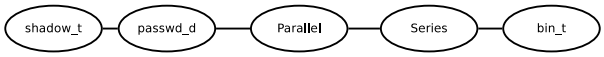


Fig. 5. The quotient graph associated to $H$, $Quot(H)$

*Definition 2:* Let $G = (V, E)$ be a graph and $\mathcal{V}$ a partition of $V$. The graph $Quot(G, \mathcal{V})$ is defined as follows:

- The vertex set is $\mathcal{V}$;
- The edge set $\mathcal{E}$ is given by:

$$\mathcal{E} = \left\{ \begin{array}{c} (V_1, V_2) \mid V_1, V_2 \in \mathcal{V} \text{ and } V_1 \neq V_2 \\ \text{and } \exists\ v_1 \in V_1, v_2 \in V_2\ (v_1, v_2) \in E \end{array} \right\}$$

Using this definition, the quotient graph in the modular decomposition context is simply $Quot(G, \mathcal{V})$ where $\mathcal{V}$ is the partition obtained from the modular decomposition tree considering the nodes at level 1.

The partition $\mathcal{V}$ obtained by selecting the nodes at the first level of the $MDTree(H)$ represented by Figure 4 is:
$\mathcal{V} =$ (`passwd_d`, `shadow_t`, (`ssh_d`, `login_d`), (`admin_d`, `user_d`), `bin_t`).

### B. Dealing with directed graphs

The graphs considered in this paper, i.e., the policy and flow graphs, are directed. The modular decomposition can also be applied to directed graphs. F. De Montgolfier [13] uses 2-structure to define the module in this context. However, we only use the undirected concepts for several reasons.

First, as shown in Section II-E, the interaction and flow graphs are used for generating the signatures. They have the same vertices, however, some edges have different directions. Applying directed modular decomposition on both graphs may generate two different modular decompositions. Consequently, the quotient graphs would be different and it would be more difficult to generate the set of signatures.

Second, the sizes of the modules in directed modular decomposition are smaller than in the undirected case, since any directed module remains a module after transformation of

the arcs into edges. Considering the initial example given in Figure 2 and its associated flow graph, the series module in the undirected decomposition is not a module in the directed case: the neighbourhood of `admin_d` and `user_d` are not the same in both graphs.

Finally, as shown at the end of this section, this action can be minimised by recovering a directed quotient graph, our process generates new dummy signatures that are useful to efficiently compress the signatures.

### C. Overall strategy

In this section, we present our global methodology. The input of our algorithm is the interaction graph $G = (V, A)$.

In order to use the modular decomposition, we consider the associated undirected graph $G' = (V, E)$ obtained by modifying any arc $(i, j)$ in $G$ into the edge $\{i, j\}$ in $G'$ and $Quot(G')$ its quotient graph. Since the generation of the signatures uses a directed graph, we compute the directed quotient graph $Quot(G, \mathcal{V})$ using the partition $\mathcal{V}$. $\mathcal{V}$ is generated by the computation of $Quot(G')$, i.e., a module of $Quot(G')$ is considered as a module of $Quot(G, \mathcal{V})$. The main difficulty is to compute the useful quotient graph depending on the source and target security contexts. Figure 6 represents the quotient graph obtained from the directed graph $G$ in Figure 2.

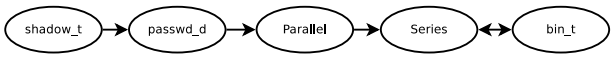


Fig. 6. The directed quotient graph associated to the flow graph Figure 2

In Section V, we show that real graphs in security domain contain several modules. They are mainly parallel modules. We show in the following section how to exploit this property for the computation of compressed signatures.

### D. Computing the quotient graph for one pair source/target

In this section, we detail the extraction of the compacted signatures from the original graph. As described in Section II-E, the signatures are computed as the set of the simple paths from the source context to the target context. Using the modular decomposition, we compute the new signatures using the quotient graph. However, the source (respectively, target) context may be contained in modules. Thus, the endpoints of the paths should be considered as a module by themselves, breaking the modules in which they are contained into smaller modules. For this, we can see that the entire module can be removed and all the elements inside considered as trivial modules. However, as shown in the Algorithm 1 some submodules can be preserved.

Note that a node appears in the quotient graph if it is a direct child of the root of the modular decomposition tree, i.e., if it appears as a singleton in the corresponding partition. The aim of the algorithm is indeed to find a partition where both endpoints are singletons while keeping the largest number of modules. We called this partition $\mathcal{P}_{s,t}$ where $s$ is the source and $t$ the target. Thus, to extract a node from a module we need to transform the modular decomposition tree such that the endpoints are direct children of the root and the tree remains a modular decomposition tree. Algorithm 1 solves this problem in the following way. Every node in the path from the root to the extracted node is a module. We can separate these modules into two categories. The first contains modules with more than two children in the tree. The second contains modules with exactly two children. Modules of the first category are put as a direct child of the root and they keep all their children but the child included in the path. Modules of the second category are removed from the tree and every child is put as a direct child of the root. Finally, the node that we want to extract is put as a direct child of the root. Special care has to be taken for prime modules. Indeed, a parallel (respectively, series) module having more than two elements can be decomposed in a parallel (respectively, series) module having exactly two elements, one node and a parallel (respectively, series) module that contained the remaining nodes. This property is not preserved for prime modules and a prime should be totally broken and its children must be put at the root level. The partition $\mathcal{P}_{s,t}$ is obtained by applying Algorithm 1 twice, with $s$ and $t$.

**Algorithm 1** Extraction of a node

```
Input: n the node to extract
Output: decomposition tree with the extracted node
  for each: node m in path(n) do
    removeEdge(Parent(m),m)
    if m is not Root and (ChildCount(m)<=2 or m is Prime)
    then
      for each: node o in child(m) do
        removeEdge(m,o)
        addEdge(Root,o)
      end for
      removeNode(m)
    else
      addEdge(Root,m)
    end if
  end for
```

Note that Algorithm 1 is very efficient since the modifications of the tree are located along the path between the root and the leaf corresponding to the node to extract. Thus, the worst case complexity is linear in the size of the tree. Consequently, the main time consuming task remains the computation of the signatures from the quotient graph. Nevertheless, the global computation time is reduced significantly.

As an example, we apply the modular decomposition on the graph depicted in Figure 7. This undirected graph contains 11 vertices and 29 edges. The algorithm generates the modular decomposition tree of Figure 8. This tree is composed of four levels and contains one prime, two series and three parallel modules. From this tree, we generate the quotient graph, in Figure 9, that contains the nodes of the first level of the tree. Computing the compressed signatures between nodes 1 and 5, we can use this quotient graph, since these nodes are not contained in non-trivial modules. There are two

 ISBN: 978-1-61208-209-7 

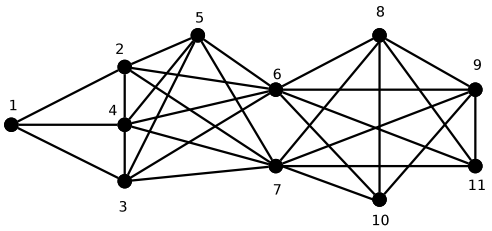


Fig. 7. Initial graph $GE$

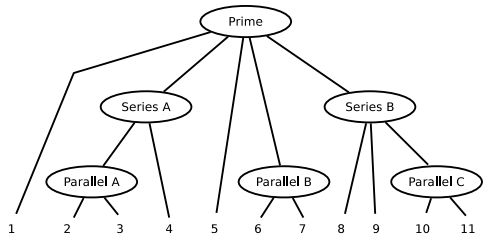


Fig. 8. Decomposition tree $MDTree(GE)$

signatures from Node 1 to Node 5 in the initial graph, there exists 2027 signatures for the same source/target pair. If the security property involves Nodes 2 and 8, this quotient graph cannot be used. Algorithm 1 computes the modified modular tree (Figure 10) and the resulting quotient graph is given in Figure 11. This graph contains 8 nodes and 15 edges. It results 50 compressed signatures instead of 2610 in the initial graph.

### *E. Reduction of the number of signatures*

In this section, we evaluate the compression rate: the number of signatures represented by a **modular signature**, i.e., a signature that contains modular contexts.

We assume in this section that the compression process using the modular decomposition does not add unwanted arcs.

*Definition 3:* Let $G = (V, A)$ a directed graph and $Q = (M, A')$ a quotient graph of $G$. An **arc** $(m_1, m_2)$ **is clean** iff

$$\forall x \in m_1, y \in m_2, (x, y) \in A.$$

The **quotient graph** $Q$ **is clean** if all its arcs are clean.

A **signature** $s$ **is clean** if it only contains clean arcs.

*Lemma 1:* A compressed clean signature $s$ that contains one modular context $M$ of size $k$ represents at least $f(k)$ non-modular signatures, where $f(k)$ is the sum of:

- $k$, i.e., the size of the module, and
- the number of simple paths between two distinct elements in the module within $G_M$, i.e., considering only the paths inside the module.

*Proof:* Let $s$ be a compressed signature that contains one modular context, i.e., $s = c_1, \ldots, c_i, M, c_{i+1}, \ldots, c_l$, where $M$ is a module in $G$, $M = \{c'_1, \ldots, c'_k\}$. Using these notations, all the $c_i$ and $c'_i$ are distinct. Otherwise, there would be a loop in the signature or one of the $c_i$ would belong to the module and should have been replaced by $M$ in $s$.

From this signature we can derive those in the original graph in two different ways. First, we can replace $M$ by one of the elements of the module. This lead to $k$ signatures of the same length as $s$. Second, we can replace $M$ by a single path between two distinct elements in $M$, leading to longer signatures and to the second part of the sum. ■

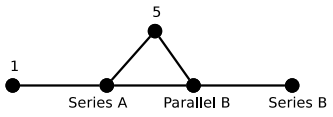


Fig. 9. Quotient graph $Quot(GE)$

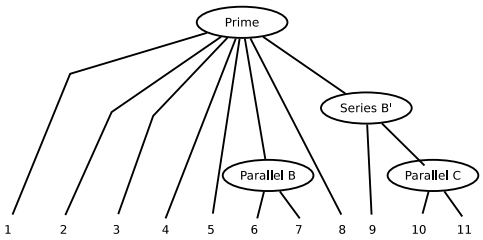


Fig. 10. Decomposition tree with source and target extract

In order to compute the computation ratio of a single signature for a general module, we need to make some complex computations. However, for parallel and series *pure* modules, this can be solved easily. We say that a module is *pure* if all its children in the modular decomposition tree are leaves.

*Consequence 1:* A compressed clean signature $s$ that contains one modular *pure* parallel context $M$ of size $k$ represents at least $k$ signatures in the original policy.

*Proof:* In this case, no path between two distinct elements exist within the module. ■

*Consequence 2:* A compressed clean signature $s$ that contains one modular *pure* series context $M$ of size $k$ represents at least $N$ signatures in the original policy, where

$$N = \sum_{i=0}^{k-1} \prod_{j=0}^{i} (k - j)$$

*Proof:* In this case, the nodes contained in $M$ form a clique. Then the number of simple paths between two elements of $M$ is at least the number of paths inside the clique, including the paths of length 0. Defining a simple path of length $i$ consists in choosing the first element within $k$, then the second within $(k-1)$, and so on. Thus, the number of simple paths of length $i$ is $\prod_{j=0}^{i}(k-j)$. By summing all these terms, we obtain the desired result. ■

If a signature contains several modules, the number of uncompressed signatures corresponds to at least the product of the number of compressed signatures for each module.

Note that a compressed non-clean signature represents some signatures that are not possible in the original graph. These unrealistic signatures are not significant for the detection process. Indeed, such uncompressed signature cannot be activated since some transitions are not allowed by the MAC policy (e.g., SELinux) of the system.

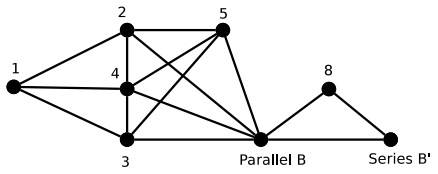


Fig. 11. Quotient graph $Quot(GE, \mathcal{P}_{2,8})$

| | Nb sig | $\rho$ | Min $\rho$ | Max $\rho$ |
|---|---|---|---|---|
| Uncompressed | 103411 | / | / | / |
| Compressed without loop | 1436 | 98.6% | 91.3% | 99.9% |
| Compressed with loops on modules | 5780 | 94.4% | 81.7% | 99.7% |

TABLE I
COMPRESSION RATIOS FOR THE EXAMPLE GRAPH

```
p1: user_u:user_r:user_t --> system_u:object_r:shadow_t
p2: user_u:user_r:user_t --> system_u:object_r:etc_t
p3: user_u:user_r:user_t --> user_u:object_r:user_tmp_t
p4: system_u:object_r:shadow_t --> user_u:user_r:user_t
p5: system_u:object_r:etc_t --> user_u:user_r:user_t
p6: user_u:object_r:user_tmp_t --> user_u:user_r:user_t
```

TABLE II
AUDITED PAIRS

| | p1 | p2 | p3 | p4 | p5 | p6 | Total |
|---|---|---|---|---|---|---|---|
| Uncompressed | 1 | 2 | 14006 | 85510 | 42756 | 10238 | 152513 |
| Compressed | 1 | 2 | 477 | 4026 | 2014 | 350 | 6870 |
| $\rho$ | 0% | 0% | 96.6% | 95.3% | 95.3% | 96.6% | 95.5% |

TABLE III
COMPRESSION RATIOS FOR THE FLOW GRAPH

## V. EXPERIMENTS

The experiments have been performed on one example graph and two graphs generated from a realistic security policy. The following sections present the structure of the studied graph and the results of our experiments for each studied case. In order to evaluate the efficiency of our method, we used the following compression ratio:

$$\rho = 1 - \frac{\#\text{compressed signatures}}{\#\text{uncompressed signatures}} \quad (1)$$

### A. Results on the example graph

The example graph (Figure 7) contains 11 nodes. This example has no application in the security domain and comes from papers dealing with modular decomposition theory [10]. However, it contains all the types of modules (Figure 8) and appears to be important in order to validate the global strategy for computing the signatures and compressed signatures.

We compute the number of signatures generated by the application of the modular decomposition for each source/-target pair. The results, presented in Table I, show that the compression is very high. Note that a compressed signature represents on average more than 70 uncompressed signatures.

We also consider more complex signatures: we allow the signature to contain loops for modular nodes. These signatures take into account the fact that a module contains several nodes. However, we use the following rule: a signature cannot contains more than $k$ occurrences of a same module if the size of this module is $k$. Even in this case, the compression ratio remains high. This investigation shows that the modular decomposition strategy has a great impact even if the sizes of the modules are not large and for any pair of source/target nodes.

Since for each pair, our strategy computes a new graph on which the signatures are generated. It appears that the graphs contain between 5 and 9 nodes.

### B. Results on a global SELinux policy

In order to evaluate the efficiency of our method on a real case, we study the following flow graph. It represents the possible information flows in a gateway using a Gentoo Linux distribution studied in [3]. It contains 43 nodes and 163 arcs. The decomposition tree has a special structure: it only contains three levels. The root is a prime node; at level 1, there exists 10 parallel modules and 16 leaves and the last level contains 27 leaves. Each parallel contains from 2 to 5 nodes. From a security point of view, each parallel has some semantic consistency. For example, one parallel contains `system_u:object_r:shadow_t`, `user_u:object_r:shadow_t` and `root:object_r:shadow_t`.

We compute the number of signatures generated on 6 source/target pairs given in Table II. The graph and the pairs were used by an instance of PIGA-IDS deployed on the gateway of a honey-pot [3]. The aim of this honey-pot was to understand the behaviour of any user considered *de facto* as an attacker. The first pair (p1) in Table II would detect all the attempts of changing the password of the user, whereas p4 detect all the attempts of the user to obtain information from the shadow file. All the pairs are using the context `user_u:user_r:user_t`, thus there are only four contexts used as endpoints. From these four contexts, only the context `system_u:object_r:shadow_t` is contained in a parallel module. Thus, all the pairs p2, p3, p5 and p6 will use the basic quotient graph for the computation of the compressed signatures, the two remaining ones use a quotient graph having 27 nodes and 91 edges. The quotient graphs are clean (see Definition 3). This implies that there are only clean signatures generated by the compression process.

Table III shows the number of signatures generated with and without using the modular decomposition process. This table also gives the compression ratio of the application of our method. A further analysis of the distribution of the compression ratios shows that a modular signature compresses between 2 and 717 non-modular signatures. Moreover, the number of modules in a signature varies between 1 and 3, some modules cannot been reached by the source element.

### C. Results on the transition policy

As described in Section II-D, from the interaction graph, we can derive several others by filtering the labels of the arcs. The transition graph is obtained by removing the arcs that are

not labelled by a transition and removing the isolated nodes. This section provides a study of the transition graph obtained from an interaction graph having 577 nodes.

This transition graph has 381 nodes and 21074 arcs (density 15%). The application of the modular decomposition generates a tree with a series module as root. This tree is deeper and more complex than the modular decomposition tree of the flow graph. At level 1, the leaves represent contexts that have some connection with all the other nodes. They are of the form `*:sysadm_r:sysadm_t` and they are related to the administrator of the system that has all the rights. Then the quotient graph is very simple and contains 5 nodes and 20 arcs corresponding to a complete graph. Note that a prime module at level 4 contains 108 of the 112 nodes at the level 5. The decomposition tree present another characteristic: the series modules contain from 3 to 7 nodes. The nodes contained in the same module have some similarities in their label. For example, all the contexts `*:*:passwd_t` form one series module of size 7. It shows that the modular decomposition reveals some logical coherency of the transition graph.

We compute the number of signatures from `system_u:system_r:sshd_t` to `user_u:user_r:user_t` on the transition graph using the modular decomposition. This computation defines all the possibilities of connections from `ssh` to a user account. We obtain 3546 signatures by limiting their length to 4. The corresponding quotient graph contains 115 nodes and 1515 arcs, 92 of them being not clean. In the original graph, it was not possible to compute all the paths of length 4 using our generation program due to memory overflow. This is a consequence of the search of the paths in the series modules that generate many signatures, as shown in Consequence 2. In this example, it was not possible to compute all the compressed signatures of length 5. To complete our experiment, we restrict the length of the paths to 3, we obtain 190 compressed signatures and 1571 uncompressed signatures, leading to a compression ratio of $\rho = 87.9\%$.

We also made the same experiment restricted to the nodes included in the prime module. In this case, the initial graph has 350 nodes and 5930 arcs, while the quotient graph used for the computation of the signatures has 108 nodes and 506 arcs, all of them being clean. The computation of the compressed signatures have been performed up to paths of length 10 where 111,704 signatures have been found, while the uncompressed signatures have been computed up to length 4. For paths of length 4, there exists only 96 compressed signatures and the compression ratio is $99\%$.

## VI. Conclusion and future work

In this paper, we have presented a way for compressing the signatures base used in PIGA IDS. We experiment this procedure on different input graphs. These experiments show that our method is efficient for compressing signatures expressing the confidentiality property: the compression ratio is usually large, over 90%. This reveals a characteristic of the interaction graph and its derivatives. Indeed, many of the modules have a logical coherency and labels inside a same module have some similarity. In some extent, we can say that modules represent some meta-contexts and that the signatures are defined at this level.

This method can be also applied to other security properties.The study on the transition graph shows that any property implying this particular graph would lead to a very high compression ratio since all the types of modules are found and the decomposition tree is deep.

The compression technique provides another result: the expressiveness of compressed signatures is very high. Indeed, compressed signatures of small size can represent very long signatures. Furthermore, since the quotient graph is smaller than the original graph, our method can compute longer signatures than the initial computing method.

The main perspective to this work is to define the detection counterpart. Indeed, the process has to be adapted in order to deal with modules in order to eliminate the false positive induced by the compaction. The additional cost due to the compression has to be clearly evaluated before being installed in a real operating system.

## References


[1] J. Goguen and J. Meseguer, "Security policies and security models," in *IEEE Symposium on Security and Privacy*, 1982, pp. 11–20.

[2] H. Mantel, "A uniform framework for the formal specification and verification of information flow security," Ph.D. dissertation, University of Saarland, 2003.

[3] 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.

[4] S. Chari and P.-C. Cheng, "BlueBoX: A policy-driven, host-based intrusion detection system," *ACM Transactions on Information and System Security*, vol. 6, no. 2, pp. 173–200, May 2003.

[5] M. Bernaschi, E. Gabrielli, and L. Mancini, "Remus: a security-enhanced operating system," *ACM Transactions on Information and System Security*, vol. 5, no. 1, pp. 36–61, Feb. 2002.

[6] G. Portokalidis, A. Slowinska, and H. Bos, "Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation," in *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, ser. EuroSys '06. Leuven, Belgium: ACM, 2006, pp. 15–27.

[7] H. Mantel, "The framework of selective interleaving functions and the modular assembly kit," in *Proceedings of the 2005 ACM workshop on Formal methods in security engineering*, ser. FMSE '05. New York, NY, USA: ACM, 2005, pp. 53–62.

[8] Department of Defense, "Trusted Computer System Evaluation Criteria," Department of Defense, Technical Report DoD 5200.28-STD, 1985.

[9] J. Bondy and U. Murty, *Graph Theory*, ser. Graduate Texts in Mathematics. Springer, 2008, vol. 244.

[10] M. Habib and C. Paul, "A survey of the algorithmic aspects of modular decomposition," *Computer Science Review*, vol. 4, no. 1, pp. 41–59, Feb. 2010.

[11] C. Papadopoulos and C. Voglis, "Drawing graphs using modular decomposition," *Journal of Graph Algorithms and Applications*, vol. 11, no. 2, pp. 481–511, 2007.

[12] J. Gagneur, R. Krause, T. Bouwmeester, and G. Casari, "Modular decomposition of protein-protein interaction networks," *Genome Biology*, vol. 5, no. 8, p. R57, 2004.

[13] R. McConnell and F. de Montgolfier, "Linear-time modular decomposition of directed graphs," *Discrete Applied Mathematics*, vol. 145, no. 2, pp. 198–209, Jan. 2005.


 ISBN: 978-1-61208-209-7 