Compression de signatures pour PIGA IDS

Pierre Clairet, Pascal Berthomé, Jérémy Briffaut

MajecSTIC 2012
Lille, France, du 29 au 31 octobre 2012

Compression de signatures pour PIGA IDS

Pierre Clairet[1], Pascal Berthomé[1] et Jérémy Briffaut[1]

1 : ENSI de Bourges, LIFO — EA 4022, 88 Boulevard Lahitolle, 18020 Bourges Cedex - France.

Contact : {pierre.clairet,pascal.berthome,jeremy.briffaut}@ensi-bourges.fr

---

Résumé
PIGA est un outil permettant de détecter les comportements malicieux par analyse de trace système. Pour cela, il utilise des signatures représentant les comportements illicites. Celles-ci sont générées à partir d'un graphe modélisant les opérations entre les différentes entités du système et sont stockées en mémoire pendant la détection d'intrusion. Dans ce papier, nous présentons un moyen de réduire la mémoire nécessaire pour stocker les signatures tout en préservant leur qualité. La méthode présentée est basée sur la décomposition modulaire des graphes. Appliquée à une propriété de confidentialité, cette méthode divise par 20 le nombre de signatures générées.

Mots-clés : sécurité, système d'exploitation, décomposition modulaire

1. Introduction

Pour assurer la sécurité d'un système informatique, on peut utiliser des systèmes de détection d'intrusion (IDS) ou de prévention d'intrusion (IPS). Ces outils cherchent en général à préserver des propriétés de sécurité. Une intrusion dans un système d'exploitation est une action qui viole une de ces propriétés. L'intégrité et la confidentialité font partie des propriétés les plus importantes. D'autres sont définies dans la littérature et sont principalement basées sur le principe de non-interférence [6, 8].
Il existe différentes techniques de détection d'intrusion [3]. Certains outils, tels que REMUS [1] ou BlueBoX [3], utilisent une approche basée sur une politique de sécurité. Argos est basé sur des signatures générées à partir d'exemples d'attaques qui ont déjà eu lieu [11]. PIGA [2] génère des signatures à partir d'une politique décrivant des propriétés de sécurité. Ces signatures représentent tous les comportements violant ces propriétés de sécurité, ce qui permet de détecter de nouvelles attaques. Elles permettent également de représenter des comportements complexes. Par exemple, des compositions d'actions élémentaires ordonnées ou non. Dans le cas d'une propriété de confidentialité, une signature est une composition séquentielle d'actions élémentaires représentant dans sa globalité une violation de cette propriété.
Ces signatures sont générées en pré-traitrement et sont utilisées lors du processus de détection. Cependant, pour un système réel, la base de signatures est importante et demande une quantité significative de mémoire pour le processus de détection. Pour un système complet Fedora avec une interface graphique, la base de signatures fait plus de 500 Mo.
Ce papier propose une amélioration du système PIGA. Dans le but de réduire l'espace mémoire, nous proposons un moyen de compacter l'information nécessaire pour la détection d'intrusion. Ce papier se concentre sur la propriété de confidentialité car elle génère un grand nombre de signatures [2]. PIGA utilisant un graphe pour représenter le système et ses interactions, cette amélioration est basée sur la réduction de ce graphe. Pour cela, nous exploitons le fait que des nœuds différents aient le même comportement vis-à-vis du reste du graphe (modularité).
Le papier s'organise comme suit. Dans la section 2, nous présentons le problème en détail au vu du fonctionnement de PIGA. La section 3 montre comment utiliser la décomposition modulaire pour compresser la base de signatures. La section 4 montre que notre stratégie peut être utilisée pour réduire la taille de l'information stockée pour détecter les menaces pour la confidentialité sur des systèmes réels. La section 5 tire quelques conclusions et perspectives.

2. Problématique

2.1. PIGA

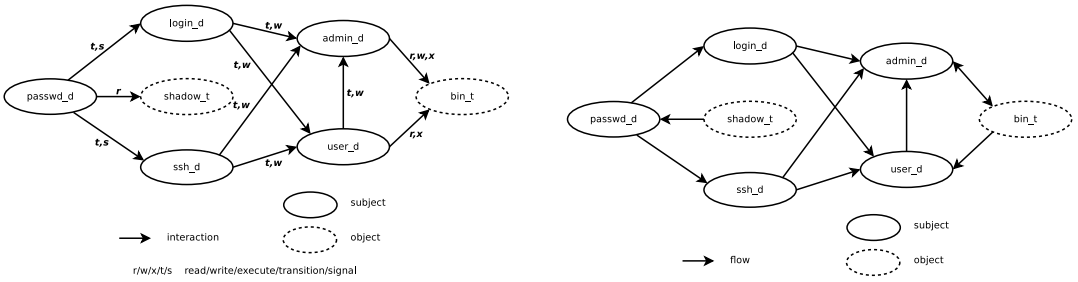
L'objectif de PIGA (Policy Interaction Graph Analysis) est de surveiller les activités système et de détecter celles violant les règles de sécurité définies par l'administrateur. Cette section propose un rappel de définitions des principaux termes de sécurité utilisés dans ce papier.

Les propriétés de sécurité permettent de définir de manière concrète la frontière entre les états *sûrs* d'un système et les états *non-sûrs*. Elles sont généralement définies par l'administrateur et constitue la politique de sécurité du système. L'*intégrité* et la *confidentialité* sont les propriétés de sécurité les plus communes [4]. La *confidentialité* permet de garantir l'absence de transfert d'information non autorisé, tandis que l'*intégrité* permet d'empêcher les modifications non souhaitées.

PIGA utilise SELinux comme système de contrôle d'accès mandataires (MAC). Il utilise donc les contextes de sécurité SELinux pour étiqueter les ressources du système (fichier, processus, ...). Plusieurs ressources peuvent appartenir au même contexte, mais une ressource n'appartient qu'à un seul contexte de sécurité. Par exemple, le contexte `system_u:object_r:ssh_port_t` regroupe l'ensemble des ports SSH des différentes interfaces du système et ces ports ne sont représentés que par ce contexte.

On appelle *interaction* une opération système entre deux contextes de sécurité. Dans une politique SELinux, seules les interactions autorisées sont décrites. Cependant cela ne garantit pas que le système reste dans un état *sûr*. En effet, une propriété de confidentialité peut être violée par une *séquence*, c'est-à-dire, une suite d'interactions. Une des caractéristiques de PIGA est d'automatiser la génération des séquences violant les propriétés de sécurité.

PIGA représente le système sous la forme d'un graphe étiqueté $G = (V, A)$, tel que V représente les contextes de sécurité et A l'ensemble des interactions entre ces contextes. Le label de l'arc allant du contexte sc_1 vers le contexte sc_2 représente l'ensemble des opérations possibles de sc_1 à sc_2. Ce graphe est appelé graphe d'interaction (figure 1a).



(a) Graphe d'interaction (b) Graphe de flux correspondant

FIGURE 1 – Modélisation d'un système d'exploitation

En filtrant la fonction d'étiquetage, on obtient, à partir du graphe d'interaction, d'autres graphes utilisés dans la génération de signatures. Ce filtrage peut supprimer des arêtes ou en inverser la direction. Dans le cas de la propriété de confidentialité, PIGA utilise le graphe de flux $FG = (V, A)$ où V représente les contextes de sécurité et A tous les transferts d'information possibles entre ces contextes. Le graphe de flux est généré en conservant les arcs impliquant des transferts d'information, principalement ceux étiquetés par des opérations de `lecture` et d'`écriture`. Une opération de lecture d'un contexte sc_1 sur un sc_2 implique un transfert d'information de sc_2 vers sc_1 : l'arc sera inversé sur le graphe de flux. Une opération d'écriture d'un contexte sc_1 sur un sc_2 implique un transfert d'information de sc_1 vers sc_2 : l'arc est conservé sur le graphe de flux. La figure 1b présente le graphe de flux déduit du graphe d'interaction de la figure 1a.

Une signature représente un ensemble d'actions, ordonnées ou non, qui viole une propriété de sécurité. Un signature correspond à une seule propriété mais une propriété peut générer plusieurs signatures. Dans le cas de PIGA, une signature représente une interaction, une séquence ou une combinaison de séquences et/ou d'interactions.

Ce papier traite de la propriété de confidentialité. En effet, les expérimentations sur PIGA [2] montrent que celle-ci est la propriété qui génère le plus grand nombre de signatures. Dans ce cas, une signature est une succession d'interactions représentant des transferts d'information possibles. Pour deux contextes sc_1 et sc_2, $sign(sc_1, sc_2)$ représente l'ensemble des signatures entre sc_1 et sc_2 et peut être calculé comme l'ensemble des chemins simples entre sc_1 et sc_2
Par exemple, l'administrateur du système correspondant au graphe de la figure 1a souhaiterait s'assurer que les fichiers du contexte `shadow_t` ne soient accessibles à aucun utilisateur (ayant le contexte `user_d`). D'après le graphe de flux de la figue 1b, de l'information de `shadow_t` peut arriver à `user_d` en seulement trois étapes sans violer la politique SELinux. En analysant le graphe de flux, on obtient 4 moyens de violer cette propriété de sécurité.

```
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
```

Tant que PIGA est actif, toutes les interactions sont analysées et le système mémorise la progression de toutes les signatures. Si une interaction complète une signature, PIGA interdit la dernière opération, s'il est en mode IPS ou avertit l'utilisateur, s'il est en mode IDS.

2.2. Définition du problème

Le calcul des signatures est fait sous forme de pré-traitement. PIGA les stocke dans le système pour les phases de détection. Des expérimentations [2] montrent que le nombre de signatures peut être élevé même pour un petit système. Le processus de détection ayant besoin de celles-ci, la mémoire nécessaire à son fonctionnement est importante (elle peut atteindre plusieurs centaines de Mo). De plus, la taille du graphe d'interaction peut rendre impossible le calcul de toutes les signatures. La solution choisie pour PIGA est de limiter la longueur des signatures générées dans la mesure où plus une signature est longue plus la probabilité qu'elle se réalise est faible.
Pour réduire la mémoire nécessaire à PIGA, nous cherchons à compresser les signatures. Actuellement, une compression simple est déjà effectuée : les signatures sont stockées en mémoire sous forme d'arbre.
Ce papier traite du problème de compression du nombre de signatures sous plusieurs contraintes. La première est de s'assurer que l'ensemble des signatures originales est inclus dans l'ensemble des signatures compressées. La deuxième contrainte est de garantir que les signatures non originales incluses dans l'ensemble compressé ne soient pas réalisables afin de ne pas générer de faux-positifs lors de la détection. Il faut par ailleurs minimiser le nombre de ces signatures *non-réalistes*. Enfin, la dernière contrainte est de conserver la génération à partir d'un graphe représentant le système.
Comme les signatures sont générées à partir du graphe de flux, si nous parvenons à réduire la taille de ce graphe, nous réduirons le nombre de signatures. Pour réduire la taille du graphe, nous regroupons les nœuds ayant le même comportement sous la forme d'un *méta-nœud*. Par exemple, pour le graphe flux de la figure 1b, les contextes `ssh_d` et `login_d` ont le même comportement vis-à-vis des autres contextes. L'idée développée dans ce papier est de regrouper ces contextes en un seul objet appelé *module*. Sur cet exemple, seules deux signatures seraient générées :

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

Cette propriété de regroupement des nœuds est un élément de base de la décomposition modulaire [7]. Pour obtenir une compression intéressante, le graphe d'interaction doit contenir suffisamment de modules. De plus, ceux-ci ne doivent pas être trop grands afin de ne pas diminuer l'efficacité du processus de détection. Le but de ce papier est d'évaluer l'efficacité de la compression par application de la décomposition modulaire sur un graphe de flux réel. La section suivante présente notre technique de compression.

3. Simplification du graphe en entrée

Dans la section 3.1, nous rappelons la définition de la décomposition modulaire des graphes non-orientés. Les graphes en entrée étant orientés, nous montrons comment nous traitons ce problème dans la section 3.2. L'algorithme initial de calcul de signatures définit une signature comme l'ensemble des chemins entre la source et la destination. Cependant, la décomposition modulaire regroupe certains nœuds sous forme de *méta-nœuds* appelé *modules*. La source et la destination peuvent donc disparaitre lors du processus. Le section 3.4 décrit un moyen de faire réapparaitre ces nœuds.

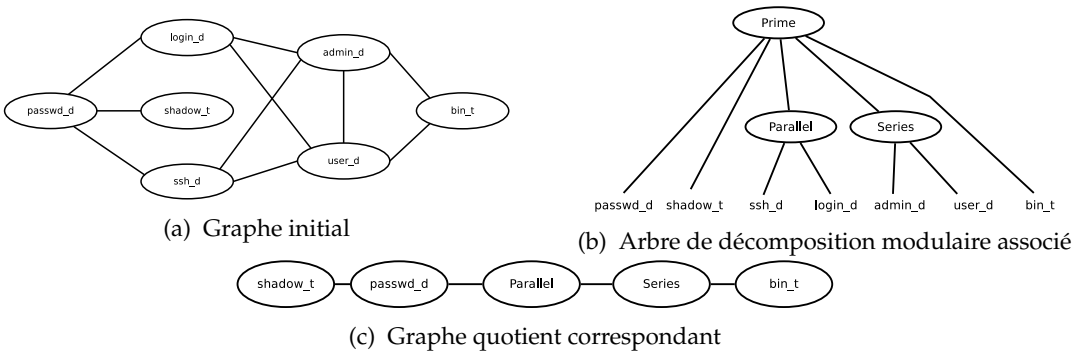
3.1. Décomposition modulaire

La décomposition modulaire [7] est un outil permettant de représenter un graphe de manière plus compacte, en groupant les nœuds ayant un comportement similaire vis-à-vis du reste du graphe. La décomposition modulaire a plusieurs applications. Par exemple, elle peut être utilisée en visualisation de graphe [10] ou en biologie [5]. Dans chacun de ces exemples, la décomposition modulaire permet de simplifier le graphe en entrée et ainsi réduire la complexité de la solution du problème. Dans ce papier, nous utilisons la décomposition modulaire afin de réduire la taille du graphe de flux utilisé pour générer les signatures.

Définition 1 *Soit* $G = (V, E)$ *un graphe non-orienté. Un* ***module*** M *de* G *est un sous-ensemble de* V *tel que :*

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

Soit G_M le sous-graphe de G tel que $G_M = (M, E')$ avec $E' = \{(x, y) \in E \mid x, y \in M\}$.
Suivant la connectivité de G_M, un module M peut être d'un des trois types suivants :
– M est un série si le complémentaire de G_M, $\overline{G_M}$, n'est pas connexe ;
– M est un parallèle si G_M n'est pas connexe ;
– M est un premier sinon.
Un module M est **fort** si pour tout autre module $M' : M \subseteq M'$, $M' \subseteq M$ ou $M \cap M' = \emptyset$. L'ensemble des modules forts est organisé sous forme d'un arbre appelé **arbre de décomposition modulaire**. La racine est le module V et les feuilles sont des modules contenant un seul nœud. Ces modules sont dit triviaux. L'arbre de décomposition modulaire du graphe G est noté $\mathrm{MDTree}(G)$.



(a) Graphe initial

(b) Arbre de décomposition modulaire associé

(c) Graphe quotient correspondant

FIGURE 2 – Éléments de la décomposition modulaire

Un **graphe quotient** est obtenu en regroupant les nœuds d'un même module en un seul nœud. Le graphe quotient de G est noté $\mathrm{Quot}(G)$. La figure 2 représente le graphe de flux de la figure 1b symétrisé (figure 2a), l'arbre de décomposition modulaire associé à ce graphe (figure 2b) et le graphe quotient généré à partir de l'arbre (figure 2c). La définition suivante étend la définition de graphe quotient à n'importe quelle partition de l'ensemble des nœuds. Cette définition est adaptable aux graphes orientés.

Définition 2 *Soit* $G = (V, E)$ *un graphe et* $\mathcal{V}$ *une partition de* V. *Le graphe* $\mathrm{Quot}(G, \mathcal{V})$ *est défini comme suit :*
– $\mathcal{V}$ *est l'ensemble des nœuds ;*
– $\mathcal{E}$ *est l'ensemble des arêtes tel que :*

$$\mathcal{E} = \{\ (V_1, V_2) \mid V_1, V_2 \in \mathcal{V}\ et\ V_1 \neq V_2\ et\ \exists\, v_1 \in V_1, v_2 \in V_2\ (v_1, v_2) \in E\ \}$$

En utilisant cette définition, le graphe quotient de la décomposition modulaire est $\mathrm{Quot}(G, \mathcal{V})$ où $\mathcal{V}$ est la partition contenant les nœuds du premier niveau de l'arbre de décomposition modulaire.

3.2. Traitement des graphes orientés

Les graphes d'interaction et de flux étudiés dans ce papier sont orientés. La décomposition modulaire peut également être appliquée aux graphes orientés. F. De Montgolfier [9] utilise les 2-structures pour définir le module orienté. Malgré cela, nous utilisons uniquement la décomposition modulaire non-orientée et cela pour plusieurs raisons.
Premièrement, les graphes d'interaction et de flux sont tous deux utilisés pour générer les signatures. Or, s'ils possèdent les mêmes nœuds, des arcs peuvent être différents. L'application de la décomposition modulaire orientée pourrait générer deux décompositions différentes. Les graphes quotient obtenus seraient donc différents ce qui rendrait la génération de signatures difficile.
Deuxièmement, étant donné qu'un module orienté reste un module après la transformation des arcs en arêtes, la taille des modules orientés est inférieure ou égale à celle des modules non-orientés. Or, des modules de taille plus grande permettent une meilleur compression.
Enfin, nous pouvons ré-orienter le graphe quotient en utilisant une version orientée de la définition 2. Ceci entraine la génération de signatures incorrectes mais permet une plus grande compression. Malgré tout, les signatures surnuméraires ne sont pas gênantes car elles ne sont peuvent pas être générées lors de la détection.

3.3. Principe général

Dans cette section, nous présentons la méthodologie utilisée pour la compression. L'entrée de notre algorithme est le graphe de flux $G = (V, A)$.
Pour pouvoir utiliser la décomposition modulaire, nous considérons le graphe non-orienté associé $G' = (V, E)$ obtenu en transformant tous les arcs de G en arêtes sur G'. Ensuite, nous calculons la décomposition modulaire sur G' pour obtenir $\mathrm{Quot}(G')$. Nous ré-orientons le graphe quotient en appliquant la partition $\mathcal{V}$ utilisée pour calculer $\mathrm{Quot}(G')$ sur G. Nous obtenons ainsi $\mathrm{Quot}(G, \mathcal{V})$ sur lequel nous calculons les signatures.
En utilisant la définition 2, nous définissons le graphe quotient du graphe orienté G comme étant $\mathrm{Quot}(G, \mathcal{P})$ où $\mathcal{P}$ est la partition définie par le premier niveau de l'arbre de décomposition modulaire du graphe non-orienté associé G'. La figure 3 représente le graphe quotient $\mathrm{Quot}(G, \mathcal{P})$ du graphe G représenté sur la figure 1b.

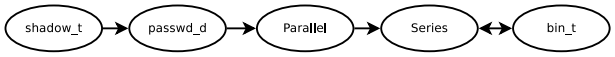


FIGURE 3 – Graphe quotient orienté associé au graphe de flux de la figure 1b

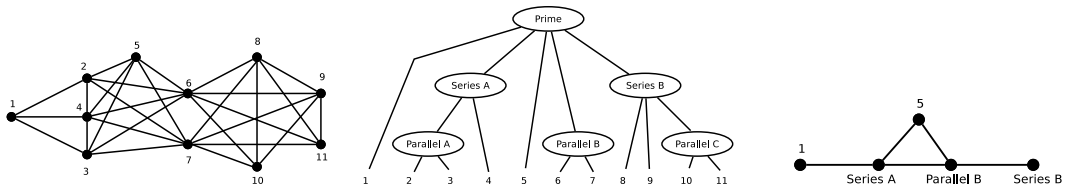
3.4. Calcul du graphe quotient pour une paire source/destination

Dans la section 2.1, nous avons vu que les signatures sont calculées comme l'ensemble des chemins entre une source et une destination. En utilisant la décomposition modulaire, nous calculons les signatures sur le graphe quotient. Or, la source (respectivement, destination) peut être contenue dans un module. Les extrémités des chemins doivent donc être considérées comme des modules, cassant les modules dans lesquels ils se trouvent en de plus petits modules. Pour cela, nous pouvons supprimer le module entier et considérer tous les nœuds qu'il contient comme des modules. Cependant, l'algorithme 1 montre qu'il est possible de conserver des sous-modules.

Algorithme 1 : Extraction d'un nœud de l'arbre de décomposition modulaire

```
   Entrées : n le nœud à extraire
   Sorties : l'arbre avec le nœud extrait
1  début
2      pour tous les nœuds m dans chemin(n) faire
3          retirerArête(Parent(m),m)
4          si m n'est pas Racine et nombreFils(m)<=2 ou m est Premier alors
5              pour tous les nœuds o dans enfants(m) faire
6                  retirerArête(m,o)
7                  ajouterArête(Racine,o)
8              retirerNœud(m)
9          sinon
10             ajouterArête(Racine,m)
11 fin
```

Un nœud apparait sur le graphe quotient s'il est un fils direct de la racine de l'arbre de décomposition modulaire, c'est-à-dire, s'il apparait comme un singleton dans la partition correspondante. L'objectif de l'algorithme 1 est donc de modifier l'arbre de décomposition modulaire de manière à ce que le nœud en entrée devienne un fils direct de la racine tout en gardant le plus grand nombre de modules possible. De plus, il faut que l'arbre reste un arbre de décomposition modulaire. Nous appliquons l'algorithme aux nœuds source s et destination d et nous obtenons ainsi une nouvelle partition appelée $\mathcal{P}_{s,d}$.
Pour illustrer la méthode, nous l'appliquons au graphe de la figure 4a. Ce graphe non-orienté contient 11 nœuds et 29 arêtes, son arbre de décomposition modulaire est présenté par la figure 4b. À partir de cet arbre, nous générons le graphe quotient (figure 4c) qui contient les nœuds du premier niveau de l'arbre. Pour calculer, les signatures entre les nœuds 1 et 5, nous pouvons utiliser ce graphe car ces deux nœuds apparaissent sur celui-ci. Nous obtenons ainsi deux signatures tandis que le même calcul sur le graphe initial en génère 2027. Pour calculer les signatures entre les nœuds 2 et 8, nous ne pouvons pas utiliser ce graphe quotient car ces nœuds n'y apparaissent pas. L'algorithme 1 appliqué successivement aux nœud 2 et 8 permet de calculer l'arbre de la figure 5a et d'en déduire le graphe quotient de la figure 5b. Ce graphe se compose de 8 nœuds et 15 arêtes. Nous calculons ainsi 50 signatures entre les nœuds 2 et 8 contre 2610 sur le graphe initial.

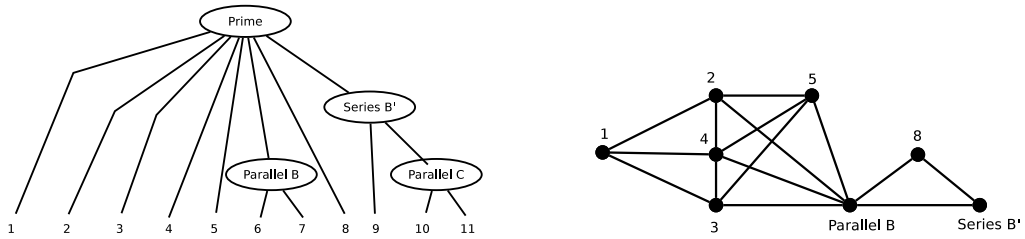


(a) Graphe initial (b) Arbre de décomposition (c) Graphe quotient

FIGURE 4 – Graphe exemple

4. Expérimentations

Nous avons expérimenté notre méthode sur un graphe exemple ainsi que sur un graphe généré à partir d'un politique existante. Pour calculer l'efficacité de notre méthode, nous utilisons le taux



(a) Arbre de décomposition modulaire modifié (b) Graphe quotient associé

FIGURE 5 – Extraction des nœuds 2 et 8 pour le graphe exemple

	Nb sig	ρ	Min ρ	Max ρ
Non-compressées	103411	/	/	/
Compressées sans boucles	1436	98.6%	91.3%	99.9%
Compressées avec des boucles sur les modules	5780	94.4%	81.7%	99.7%

TABLE 1 – Taux de compression pour le graphe exemple

de compression suivant :

$$\rho = 1 - \frac{\#\text{signatures compressées}}{\#\text{signatures non-compressées}} \tag{1}$$

4.1. Résultats sur un graphe exemple

Le graphe exemple, présenté sur la figure 4a, est composé de 11 nœuds et 29 arêtes. Ce graphe, tiré de [7], n'a aucune application en sécurité mais contient tous les types de modules.

Pour évaluer l'efficacité de notre méthode, nous calculons le nombre de signatures générées avec et sans application de la décomposition modulaire pour chaque paire source/destination possible. Les résultats sont présentés dans la table 1 et montrent que la compression obtenue est importante. Le taux de compression est calculé pour chaque paire avec deux méthodes différentes pour les signatures compressées. Premièrement, les signatures compressées sont calculées sans boucle. Deuxièmement, elles sont calculées en autorisant les boucles sur les modules en respectant la règle suivante : une signature peut contenir k fois un même module si ce module est de taille k. Pour les deux méthodes, le taux de compression obtenu est important et montre que la décomposition modulaire a un impact important même avec des modules de petite taille.

4.2. Résultats sur un graphe de flux réel

Afin d'évaluer l'efficacité de notre méthode sur un cas réel, nous l'appliquons sur un graphe de flux existant. Ce graphe, composé de 43 nœuds et 163 arcs, représente les possibles flux d'information sur une passerelle sous Gentoo [2]. L'arbre de décomposition obtenu possède une structure particulière : il n'a que trois niveaux et tous ses modules sont des parallèles. Ceux-ci contiennent entre 2 et 5 nœuds et possèdent une cohérence sémantique. Par exemple, un parallèle contient `system_u:object_r:shadow_t`, `user_u:object_r:shadow_t` et `root:object_r:shadow_t`.

Nous calculons le nombre de signatures générées pour 6 paires source/destination. Le graphe quotient obtenu est différent pour chaque paire et possède au plus 27 nœuds et 91 arcs.

La table 2 présente le nombre de signatures non-compressées et compressées pour chaque paire ainsi que le taux de compression associé. Cette expérimentation montre par ailleurs qu'une signature modulaire compresse entre 2 et 717 signatures non-modulaire et comporte entre 1 et 3 modules.

	p1	p2	p3	p4	p5	p6	Total
Non-compressées	1	2	14006	85510	42756	10238	152513
Compressées	1	2	477	4026	2014	350	6870
ρ	0%	0%	96.6%	95.3%	95.3%	96.6%	95.5%

TABLE 2 – Taux de compression pour le graphe de flux

5. Conclusion

Nous avons présenté une méthode permettant de compresser la base de signatures utilisée par PIGA-IDS. Nous avons expérimenté cette méthode sur différents graphes. Ces expérimentations montrent que notre méthode est efficace pour compresser les signatures associées à la propriété de confidentialité : le taux de compression est généralement supérieur à 90%. Cette études met en évidence une cohérence logique des modules appuyée par le fait que les nœuds appartenant au même module ont des similarités. Par extension, nous pouvons dire que les modules représentent des méta-contextes. En plus de l'efficacité en termes de compression de notre méthode, celle-ci permet de générer des signatures plus longues que la méthode de calcul initiale. En effet, de petites signatures modulaires peuvent compresser des signatures non-modulaires plus longues. De plus, le graphe quotient étant plus petit que le graphe original, nous pouvons calculer des signatures plus longues avant d'utiliser toutes les ressources du système.
La principale perspective de ces travaux est de définir l'impact de la compression sur la détection. En effet, le processus doit être adapté à l'utilisation des signatures modulaires pour éliminer les cas de faux positif induits par la compression. Le surcoût dû à la compression doit être clairement évalué avant de pouvoir utiliser notre méthode sur un système d'exploitation réel.

Bibliographie

1. M. Bernaschi, E. Gabrielli, et L.V. Mancini. Remus : a security-enhanced operating system. *ACM Transactions on Information and System Security*, 5(1) :36–61, février 2002.
2. J. Briffaut, J. Rouzaud-Cornabas, C. Toinard, et Y. Zemali. A new approach to enforce the security properties of a clustered high-interaction honeypot. In Ratan Kumar Guha et Luca Spalazzi, editors, *Workshop on Security and High Performance Computing Systems*, pages 184–192, Leipzig, Germany, June 2009. IEEE Computer Society.
3. S.N. Chari et P.-C. Cheng. Bluebox : A policy-driven, host-based intrusion detection system. *ACM Transactions on Information and System Security*, 6(2) :173–200, mai 2003.
4. Department of Defense. Trusted Computer System Evaluation Criteria. Technical Report DoD 5200.28-STD, Department of Defense, 1985.
5. J. Gagneur, R. Krause, T. Bouwmeester, et G. Casari. Modular decomposition of protein-protein interaction networks. *Genome Biology*, 5(8) :R57, 2004.
6. J.A. Goguen et J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.
7. M. Habib et C. Paul. A survey of the algorithmic aspects of modular decomposition. *Computer Science Review*, 4(1) :41–59, février 2010.
8. H. Mantel. *A Uniform Framework for the Formal Specification and Verification of Information Flow Security*. Thèse de doctorat, University of Saarland, 2003.
9. R McConnell et F de Montgolfier. Linear-time modular decomposition of directed graphs. *Discrete Applied Mathematics*, 145(2) :198–209, janvier 2005.
10. C. Papadopoulos et C. Voglis. Drawing graphs using modular decomposition. *Journal of Graph Algorithms and Applications*, 11(2) :481–511, 2007.
11. G. Portokalidis, A. Slowinska, et 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*, EuroSys '06, pages 15–27, Leuven, Belgium, 2006. ACM.